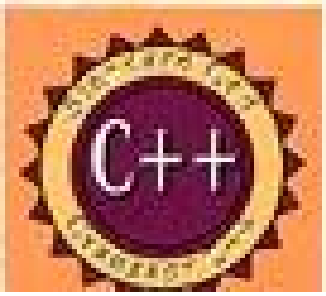
Язык программирования C++

Вводный курс

Стенли Б. Липпман
Жози Лажойе

Третье издание



Предисловие

Между выходом второго и третьего издания “C++ для начинающих” произошло довольно много событий. Одним из самых значительных стало появление международного стандарта. Он не только добавил в язык C++ новые возможности, среди которых обработка исключений, идентификация типов во время выполнения, пространство имен, встроенный булевский тип данных, новый синтаксис приведения типов, но также существенно изменил и расширил имеющиеся – шаблоны, механизм классов, поддерживающий объектную и объектно-ориентированную парадигму программирования, вложенные типы и разрешение перегруженных функций. Еще более важным событием стало включение в состав стандарта C++ обширной библиотеки, содержащей, в частности, то, что ранее называлось Standard Template Library (STL). В эту стандартную библиотеку входят новый тип `string`, последовательные и ассоциативные контейнеры, такие, как `vector`, `list`, `map`, `set`, и обширный набор обобщенных алгоритмов, которые могут применяться ко всем этим типам данных. Появилось не просто много нового материала, нуждающегося в описании, но фактически изменился сам способ мышления при программировании на C++. Короче говоря, можно считать, что C++ изобретен заново, поэтому третье издание нашей книги “C++ для начинающих” полностью переработано.

В третьем издании не только коренным образом поменялся наш подход к C++, изменились и авторы. Прежде всего, авторский коллектив удвоился и стал интернациональным, хотя корни его по-прежнему на североамериканском континенте: Стен (Stan) американец, а Жози (Josйe) канадка. Двойное авторство отражает деление сообщества программистов C++ на две части: Стен в настоящее время занимается разработкой приложений на C++ в области трехмерной графики и анимации для Walt Disney Feature Animation, а Жози принимает участие в развитии самого языка C++, являясь председателем рабочей группы по ядру языка в комитете по стандартизации и одним из разработчиков компилятора C++ в IBM Canada Laboratory.

Стен работает над C++ с 1984 года. Он был одним из членов первоначальной команды, трудившейся в Bell Laboratories под руководством Бьерна Страуструпа (Bjarne Stroustrup), изобретателя языка. Стен принимал участие в разработке cfront, оригинальной реализации C, с версии 1.1 в 1986 году до версии 3.0, и возглавлял проект при работе над версиями 2.1 и 3.0. После этого он работал под началом Страуструпа над проектом, посвященным исследованиям объектной модели программной среды разработки на C++.

Жози – член команды, работающей над компилятором C++ в IBM Canada Laboratory на протяжении восьми лет. С 1990 года она входит в состав комитета по стандартизации. Три года она была вице-президентом комитета и четыре года – председателем рабочей группы по ядру языка.

Третье издание “C++ для начинающих” существенно переработано, что отражает не только развитие и расширение языка, но и изменения во взглядах и опыте авторов книги.

Структура книги

“C++ для начинающих” содержит обстоятельное введение в международный стандарт C++. Мы включили в название книги слова “для начинающих” потому, что последовательно придерживались учебного подхода к описанию языка C++; однако название не предполагает упрощенного или облегченного изложения материала. Такие аспекты программирования, как обработка исключений, контейнерные типы, объектно-ориентированный подход и т.п.,

представлены в книге в контексте решения конкретных задач. Правила языка, например разрешение перегруженных функций или преобразования типов в объектно-ориентированном программировании, рассматриваются столь подробно, что во вводном курсе это может показаться неуместным. Но мы уверены, что такое освещение необходимо для практического применения языка. Материал книги не нужно стараться усвоить “за один проход”: мы предполагаем, что читатель будет периодически возвращаться к ранее прочитанным разделам. Если некоторые из них покажутся вам слишком трудными или просто скучными, отложите их на время. (Подозрительные разделы мы помечали значком A.)

Читатель может не знать язык C, хотя некоторое знакомство с каким-либо современным структурным языком программирования было бы полезно. Мы писали книгу, которая стала бы первым учебником по C++, а не первым учебником по программированию! Чтобы не делать предположений о начальном уровне подготовки, мы начинаем с определения базовых терминов. В первых главах описываются базовые концепции, такие, как переменные и циклы, и для некоторых читателей изложение может показаться слишком примитивным, но вскоре оно становится более углубленным.

Основное достоинство C++ заключается в том, что он поддерживает новые способы решения программистских задач. Поэтому чтобы научиться эффективно использовать C++, недостаточно просто выучить новые синтаксис и семантику. Для более глубокого усвоения в книге рассматриваются разнообразные сквозные примеры. Эти примеры используются как для того, чтобы представить разные средства языка, так и для того, чтобы объяснить, зачем эти средства нужны. Изучая возможности языка в контексте реального примера, мы понимаем, чем полезно то или иное средство, как и где его можно применить при решении задач из реальной жизни. Кроме того, на примерах проще продемонстрировать понятия языка, которые еще детально не рассматривались и излагаются лишь в последующих главах. В начальных главах примеры содержат простые варианты использования базовых понятий C++. Их цель – показать, как можно программировать на C++, не углубляясь в детали проектирования и реализации.

Главы 1 и 2 представляют собой полное введение в язык C++ и его обзор. Назначение первой части – как можно быстрее познакомить читателя с понятиями и средствами данного языка, а также основными принципами написания программ.

По окончании этой части у вас должно сложиться некоторое общее представление о возможностях C++, но вместе с тем вполне может остаться ощущение, что вы *совсем ничего толком не понимаете*. Все нормально: упорядочению ваших знаний как раз и посвящены остальные части книги!

В главе 1 представлены базовые элементы языка: встроенные типы данных, переменные, выражения, инструкции и функции. Мы увидим минимальную законченную программу на C++, обсудим вопросы компиляции, коснемся препроцессора и поддержки ввода/вывода. В этой главе читатель найдет ряд простых, но законченных C++ программ, которые можно откомпилировать и выполнить. Глава 2 посвящена механизму классов и тому, как с его помощью поддержаны парадигмы объектного и объектно-ориентированного программирования. Оба эти подхода иллюстрируются развитием реализации массива как абстрактного типа. Кроме того, приводится краткая информация о шаблонах, пространствах имен, обработке исключений и о поддержке стандартной библиотекой общих контейнерных типов и методов обобщенного (generic) программирования. Материал в этой главе излагается весьма стремительно, и потому некоторым читателям она может показаться трудной. Мы

рекомендуем таким читателям просмотреть вторую главу “по диагонали” и вернуться к ней впоследствии.

Фундаментальной особенностью C++ является возможность расширять язык, определяя новые типы данных, которые могут использоваться с тем же удобством и гибкостью, что и встроенные. Первым шагом к овладению этим искусством является знание базового языка. Часть II (главы 3-6) посвящена рассмотрению языка на этом уровне.

В главе 3 представлены встроенные и составные типы, предопределенные в языке, а также типы `string`, `complex` и `vector` из стандартной библиотеки C++. Эти типы составляют основные “кирпичики”, из которых строятся все программы. В главе 4 детально освещаются выражения языка – арифметические, условные, присваивания. Инструкции языка, которые являются мельчайшими независимыми единицами C++ программы, представлены в главе 5. Контейнерные типы данных стали предметом главы 6. Вместо простого перечисления совокупности поддерживаемых ими операций, мы иллюстрируем операции на примере построения системы текстового поиска.

Главы 7-12 (часть III) посвящены *процедурно-ориентированному* программированию на C++. В главе 7 представлен механизм функций. Функция инкапсулирует набор операций, составляющих единую задачу, как, например, `print()`. (Круглые скобки после имени говорят о том, что мы имеем дело с функцией.) Такие понятия, как область видимости и время жизни переменных, рассматриваются в главе 8. Обзор механизма функций продолжен в главе 9: речь пойдет о перегрузке функций, которая позволяет присвоить одно и то же имя нескольким функциям, выполняющим похожие, но по-разному реализованные операции. Например, можно определить целый набор функций `print()` для печати данных разных типов. В главе 10 представлено понятие шаблона функции и приведены примеры его использования. Шаблон функции предназначен для автоматической генерации потенциально бесконечного множества экземпляров функций, отличающихся только типами данных.

C++ поддерживает обработку исключений. Об исключении говорят, когда в программе возникает нестандартная ситуация, такая, например, как нехватка свободной памяти. В том месте программы, где это происходит, *возбуждается* исключение, то есть о проблеме ставится в известность вызывающая программа. Какая-то другая функция в программе должна *обработать* исключение, то есть как-то отреагировать на него. Материал об исключениях разбит на две части. В главе 11 описан основной синтаксис и приведен простой пример, иллюстрирующий возбуждение и обработку исключений типа класса. Поскольку реальные исключения в программах обычно являются объектами некоторой иерархии классов, то мы вернемся к этому вопросу в главе 19, после того как узнаем, что такое объектно-ориентированное программирование.

В главе 12 представлена обширная коллекция обобщенных алгоритмов стандартной библиотеки и способы их применения к контейнерным типам из главы 6, а также к массивам встроенных типов. Эта глава начинается разбором примера построения программы с использованием обобщенных алгоритмов. Итераторы, введенные в главе 6, обсуждаются более детально в главе 12, поскольку именно они являются связующим звеном между обобщенными алгоритмами и контейнерными типами. Также мы вводим и иллюстрируем на примерах понятие объекта-функции. Объекты-функции позволяют задавать альтернативную семантику операций, используемых в обобщенных алгоритмах, – скажем, операций сравнения на равенство или по величине. Детальное описание самих алгоритмов и примеры их использования приводятся в приложении.

Главы 13-16 (часть IV) посвящены *объектному* программированию, то есть использованию механизма классов для создания абстрактных типов данных. С помощью типов данных, описывающих конкретную предметную область, язык C++ позволяет программистам сосредоточиться на решении основной задачи и тратить меньше усилий на второстепенные. Фундаментальные для приложения типы данных могут быть реализованы один раз и использованы многократно, что дает программисту возможность не думать о деталях реализации главной задачи. Инкапсуляция данных значительно упрощает последующее сопровождение и модификацию программы.

В главе 13 основное внимание мы уделим общим вопросам механизма классов: как определить класс, что такое *сокрытие информации* (разделение открытого интерфейса и скрытой реализации), как определять экземпляры класса и манипулировать ими. Мы также коснемся областей видимости класса, вложенных классов и классов как членов пространства имен.

В главе 14 детально исследуются средства, имеющиеся в C++ для инициализации и уничтожения объектов класса и для присваивания им значений. Для этих целей служат специальные функции-члены, называемые *конструкторами*, *деструкторами* и копирующими *операторами присваивания*. Мы рассмотрим вопрос о почленной инициализации и копировании, а также специальную оптимизацию для этого случая, которая получила название *именованное возвращаемое значение*.

В главе 15 мы рассмотрим перегрузку операторов применительно к классам. Сначала мы остановимся на общих понятиях и вопросах проектирования, а затем перейдем к рассмотрению конкретных операторов, таких, как присваивание, доступ по индексу, вызов функции, а также операторов `new` и `delete`, специфичных для классов.

Будет представлено понятие *дружественного класса*, имеющего особые права доступа, и объяснено, зачем нужны *друзья*. Будут рассмотрены и определенные пользователями преобразования типов, стоящие за ними концепции и примеры использования. Кроме того, приводятся правила разрешения функций при перегрузке, иллюстрируемые примерами программного кода.

Шаблоны классов – тема главы 16. Шаблон класса можно рассматривать как алгоритм создания экземпляра класса, в котором параметры шаблона подлежат замене на конкретные значения типов или констант. Скажем, в шаблоне класса `vector` параметризован тип его элементов. В классе для представления некоторого буфера можно параметризовать не только тип размещаемых элементов, но и размер самого буфера. При разработке сложных механизмов, например в области распределенной обработки данных, могут быть параметризованы практически все интерфейсы: межпроцессной коммуникации, адресации, синхронизации. В главе 16 мы расскажем, как определить шаблон класса, как создать экземпляр класса, подставляя в шаблон конкретные значения, как определить члены шаблона класса (функции-члены, статические члены и вложенные типы) и как следует организовывать программу, в которой используются шаблоны классов. Заканчивается эта глава содержательным примером шаблона класса.

Объектно-ориентированному программированию (ООП) и его поддержке в C++ посвящены главы 17-20 (часть IV). В главе 17 описываются средства поддержки базовых концепций ООП – наследования и позднего связывания. В ООП между классами, имеющими общие черты поведения, устанавливаются отношения родитель/потомок (или тип/подтип). Вместо того чтобы повторно реализовывать общие характеристики, класс-потомок может унаследовать их от класса-родителя. В класс-потомок (подтип) следует добавить только те

детали, которые отличают его от родителя. Например, мы можем определить родительский класс `Employee` (работник) и двух его потомков: `TemporaryEmpl` (временный работник) и `Manager` (начальник), которые наследуют все поведение `Employee`. В них самих реализованы только специфичные для подтипа особенности. Второй аспект ООП, *полиморфизм*, позволяет родительскому классу представлять любого из своих наследников. Скажем, класс `Employee` может адресовать не только объект своего типа, но и объект типа `TemporaryEmpl` или `Manager`. Позднее связывание – это способность разрешения операций во время выполнения, то есть выбора нужной операции в зависимости от реального типа объекта. В C++ это реализуется с помощью механизма виртуальных функций.

Итак, в главе 17 представлены базовые черты ООП. В ней мы продолжим начатую в главе 6 работу над системой текстового поиска – спроектируем и реализуем иерархию классов запросов `Query`.

В главе 18 разбираются более сложные случаи наследования – множественное и виртуальное. Шаблон класса из главы 16 получает дальнейшее развитие и становится трехуровневой иерархией с множественным и виртуальным наследованием.

В главе 19 представлено понятие идентификации типа во время выполнения (RTTI – run time type identification). RTTI позволяет программе запросить у полиморфного объекта класса информацию о его типе во время выполнения. Например, мы можем спросить у объекта `Employee`, действительно ли он представляет собой объект типа `Manager`. Кроме того, в главе 19 мы вернемся к исключениям и рассмотрим иерархию классов исключений стандартной библиотеки, приводя примеры построения и использования своей собственной иерархии классов исключений. В этой главе рассматривается также вопрос о разрешении перегруженных функций в случае наследования классов.

В главе 20 подробно рассматривается использование библиотеки ввода/вывода `iostream`. Здесь мы на примерах покажем основные возможности ввода и вывода, расскажем, как определить свои операторы ввода и вывода для класса, как проверять состояние потока и изменять его, как форматировать данные. Библиотека ввода/вывода представляет собой иерархию классов с множественным и виртуальным наследованием.

Завершается книга приложением, где все обобщенные алгоритмы приведены в алфавитном порядке, с примерами их использования.

При написании книги зачастую приходится оставлять в стороне множество вопросов, которые представляются не менее важными, чем вошедшие в книгу. Отдельные аспекты языка – детальное описание того, как работают конструкторы, в каких случаях создаются временные объекты, общие вопросы эффективности – не вписывались во вводный курс. Однако эти аспекты имеют огромное значение при проектировании реальных приложений. Перед тем как взяться за “C++ для начинающих”, Стен написал книгу “Inside the C++ Object Model” [LIPPMAN96a], в которой освещаются именно эти вопросы. В тех местах “C++ для начинающих”, где читателю может потребоваться более детальная информация, даются ссылки на разделы указанной книги.

Некоторые части стандартной библиотеки C++ были сознательно исключены из рассмотрения, в частности поддержка национальных языков и численные методы. Стандартная библиотека C++ очень обширна, и все ее аспекты невозможно осветить в одном учебнике. Материал по отсутствующим вопросам вы можете найти в книгах, приведенных в списке литературы ([MUSSER96] и [STRUOSTRUP97u]). Наверняка вскоре выйдет еще немало книг, освещающих различные аспекты стандартной библиотеки C++.

Изменения в третьем издании

Все изменения можно разбить на четыре основные категории:

материал, посвященный нововведениям языка: обработке исключений, идентификации типа во время выполнения, пространству имен, встроенному типу bool, новому синтаксису приведения типов;

материал, посвященный стандартной библиотеке C++, в том числе типам complex, string, auto_ptr, pair, последовательным и ассоциативным контейнерам (в основном это list, vector, map и set) и обобщенным алгоритмам;

коррективы в старом тексте, отражающие улучшения, расширения и изменения, которые новый стандарт C++ привнес в существовавшие ранее средства языка. Примером улучшения может служить использование предваряющих объявлений для вложенных типов, ранее отсутствовавшая. В качестве примера изменения можно привести возможность для экземпляра виртуальной функции производного класса возвращать тип, производный от типа значения, возвращаемого экземпляром той же функции из базового класса. Это изменение поддерживает операцию с классами, которую иногда называют *клонированием* или *фабрикацией классов* (виртуальная функция clone() иллюстрируется в разделе 17.5.7). Пример расширения языка – возможность явно специализировать один или более параметров-типов для шаблонов функций (на самом деле, весь механизм шаблонов был радикально расширен – настолько, что его можно назвать новым средством языка!);

изменения в подходе к использованию большинства продвинутых средств языка – шаблонов и классов. Стен считает, что его переход из сравнительно узкого круга разработчиков языка C++ в широкий круг пользователей позволил ему глубже понять проблемы последних. Соответственно в этом издании мы уделили большее внимание концепциям, которые стояли за появлением того или иного средства языка, тому, как лучше его применять и как избежать подводных камней.

Будущее C++

Во время публикации книги комитет по стандартизации C++ ISO/ANSI закончил техническую работу по подготовке первого международного стандарта C++. Стандарт опубликован Международным комитетом по стандартизации (ISO) летом 1998 года.

Реализации C++, поддерживающие стандарт, должны появиться вскоре после его публикации. Есть надежда, что после публикации стандарта изменения в C++ перестанут быть столь радикальными. Такая стабильность позволит создать сложные библиотеки, написанные на стандартном C++ и направленные на решение различных промышленных задач. Таким образом, основной рост в мире C++ ожидается в сфере создания библиотек.

После публикации стандарта комиссия тем не менее продолжает свою работу, хотя и не так интенсивно. Разбираются поступающие от пользователей вопросы по интерпретации тех или иных особенностей языка. Это приводит к небольшим исправлениям и уточнениям стандарта C++. При необходимости международный стандарт будет пересматриваться каждые пять лет, чтобы учесть изменения в технологиях и нужды отрасли.

Что будет через пять лет после публикации стандарта, пока неизвестно. Возможно, новые компоненты из прикладных библиотек войдут в стандартную библиотеку C++. Но сейчас, после окончания работы комиссии, судьба C++ оказывается в руках его пользователей.

Благодарности

Особые благодарности, как обычно, мы выражаем Бьерну Страуструпу за прекрасный язык, который он подарил нам, и за то внимание, которое он оказывал ему все эти годы. Особые благодарности членам комитета по стандартизации C++ за их самоотверженность и упорную работу (часто безвозмездную) и за огромный вклад в появление Стандарта C++.

На разных стадиях работы над рукописью многие люди вносили различные полезные замечания: Пол Эбрахамс (Paul Abrahams), Майкл Болл (Michael Ball), Стивен Эдвардс (Stephen Edwards), Кэй Хорстманн (Cay Horstmann), Брайан Керниган (Brian Kernighan), Том Лайонс (Tom Lyons), Роберт Мюррей (Robert Murray), Эд Шейбель (Ed Scheibel), Рой Тэрнер (Roy Turner), Йон Вада (Jon Wada). Особо нам хочется поблагодарить Майкла Болла за важные комментарии и поддержку. Мы благодарим Кловис Тондо (Clovis Tondo) и Брюса Леюнга (Bruce Leung) за вдумчивую рецензию.

Стен выражает особо теплую благодарность Ши-Чюань Хуань (Shyh-Chyuan Huang) и Джинко Гото (Jinko Gotoh) за их помощь в работе над рассказом о Жар-Птице (Firebird), Иона Ваду и, конечно, Джози.

Джози благодарит Габби Зильберман (Gabby Silbermarr), Карен Беннет (Karen Bennet), а также команду Центра углубленных исследований (Centre for Advanced Studies) за поддержку во время написания книги. И выражает огромную благодарность Стену за привлечение к работе над книгой.

Мы оба хотим поблагодарить замечательный редакторский коллектив за их упорную работу и безграничное терпение: Дебби Лафферти (Debbie Lafferty), которая не оставляла вниманием эту книгу с самого первого издания, Майка Хендриксона (Mike Hendrickson) и Джона Фуллера (John Fuller). Компания Big Purple Company проделала замечательную работу по набору книги. Иллюстрация в разделе 6.1 принадлежит Елене Дрискилл (Elena Driskill). Мы благодарим ее за разрешение перепечатки.

Благодарности во втором издании

Эта книга явилась результатом работы множества остающихся за сценой людей, помогавших автору. Наиболее сердечные благодарности мы приносим Барбаре Му (Barbara Moo). Ее поддержка, советы, внимательное чтение бесчисленных черновиков книги просто неоценимы. Особые благодарности Бьярну Страуструпу за постоянную помощь и поддержку и за прекрасный язык, который он подарил нам, а также Стивену Дьюхерсту (Stephen Dewhurst), который так много помогал мне при освоении C++, и Ненси Уилкинсон (Nancy Wilkinson) – коллеге по работе над cfront.

Дэг Брюк (Dag Bruck), Мартин Кэрролл (Martin Carroll), Уильям Хопкинс (William Hopkins), Брайан Керниган (Brian Kernighan), Эндрю Кениг (Andrew Koenig), Алексис Лейтон (Alexis Layton) и Барбара Му (Barbara Moo) помогали нам особо ценными замечаниями. Их рецензии значительно улучшили качество книги. Энди Бейли (Andy Baily), Фил Браун (Phil Brown), Джеймс Коплиен (James Coplien), Элизабет Флэнаган (Elizabeth Flanagan), Дэвид Джордан (David Jordan), Дон Кретч (Don Kretsch), Крейг Рубин (Craig Rubin), Джонатан Шопиро (Jonathan Shopiro), Джуди Уорд (Judy Ward), Ненси Уилкинсон (Nancy Wilkinson) и Клей Уилсон (Clay Wilson) просмотрели множество черновиков книги и дали много полезных комментариев. Дэвид Проссер (David Prosser) прояснил множество вопросов, касающихся ANSI C.

Джерри Шварц (Jerry Schwarz), автор библиотеки `iostream`, обеспечил нас оригинальной документацией, которая легла в основу Приложения A (глава 20 в третьем издании). Мы высоко оцениваем его замечания к этому Приложению. Мы благодарим всех остальных

членов команды, работавшей на версией 3.0: Лауру Ивс (Laura Eaves), Джорджа Логотетиса (George Logothetis), Джуди Уорд (Judy Ward) и Ненси Уилкинсон (Nancy Wilkinson).

Джеймс Эдкок (James Adcock), Стивен Белловин (Steven Bellovin), Йон Форрест (Jon Forrest), Морис Эрлих (Maurice Herlihy), Норман Керт (Norman Kerth), Даррелл Лонг (Darrell Long), Виктор Миленкович (Victor Milenkovic) и Джастин Смит (Justin Smith) рецензировали книгу для издательства Addison-Wesley.

Дэвид Беккердорф (David Beckedorff), Дэг Брюк (Dag Bruck), Джон Элбридж (John Eldridge), Джим Хьюмелсин (Jim Humelsine), Дэйв Джордан (Dave Jordan), Эми Клейнман (Ami Kleinman), Эндрю Кениг (Andrew Koenig), Тим О'Конски (Tim O'Konski), Кловис Тондо (Clovis Tondo) и Стив Виноски (Steve Vinoski) указали на ошибки в первом издании.

Я выражаю глубокую благодарность Брайану Кернигану (Brian Kernighan) и Эндрю Кенигу (Andrew Koenig) за программные средства для типографского набора текста.

Список литературы

Следующие работы либо оказали большое влияние на написание данной книги, либо представляют ценный материал по C++, который мы рекомендуем читателю.

[BOOCH94] Booch, Grady, *Object-Oriented Analysis and Design,* Benjamin/Cummings. Redwood City, CA (1994) ISBN 0-8053-5340-2.

[GAMMA95] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns, Addison Wesley Longman, Inc., Reading, MA (1995) ISBN 0-201-63361-2.

[GHEZZI97] Ghezzi, Carlo, and Mehdi Jazayeri, Programming Language Concepts, 3rd Edition, John Wiley and Sons, New York, NY (1997) ISBN 0-471-10426-4.

[HARBISON88] Samuel P. Harbison and Guy L. Steele, Jr., C: A Reference Manual, 3rd Edition, Prentice-Hall, Englewood Cliffs, NJ (1988) ISBN 0-13-110933-2.

[ISO-C++97] Draft Proposed International Standard for Information Systems — Programming Language C++ – Final Draft (FDIS) 14882.

[KERNIGHAN88] Kernighan, Brian W.I. and Dennis M. Ritchie, The C Programming Language, Prentice-Hall, Englewood Cliffs, NJ (1988) ISBN 0-13-110362-8.

[KOENIG97] Koenig, Andrew, and Barbara Moo, Ruminations on C++, Addison Wesley Longman, Inc., Reading, MA (1997) ISBN 0-201-42339-1.

[LIPPMAN91] Lippman, Stanley, C++ Primer, 2nd Edition, Addison Wesley Longman, Inc., Reading, MA (1991) ISBN 0-201-54848-8.

[LIPPMAN96a] Lippman, Stanley, Inside the C++ Object Model, Addison Wesley Longman, Inc., Reading, MA (1996) ISBN 0-201-83454-5.

[LIPPMAN96b] Lippman, Stanley, Editor, C++ Gems, a SIGS Books imprint, Cambridge University Press, Cambridge, England (1996) ISBN 0-13570581-9.

[MEYERS98] Movers, Scott, Effective C++, 2nd Edition, Addison Wesley Longman, Inc., Reading, MA (1998) ISBN 0-201-92488-9.

[MEYERS96] Meyers, Scott, More Effective C++, Addison Wesley Longman, Inc., Reading, MA (1996) ISBN 0-201-63371-X.

[MURRAY93] Murray Robert B., C++ Strategies and Tactics, Addison Wesley Longman, Inc., Reading, MA (1993) ISBN 0-201-56382-7.

[MUSSER96] Musser, David R., and Atui Saint, STL Tutorial and Reference Guide, Addison Wesley Longman, Inc., Reading, MA (1996) ISBN 0-201-63398-1.

[NACKMAN94] Barton, John J., and Lee R. Nackman, Scientific and Engineering C++, An Introduction with Advanced Techniques and Examples, Addison Wesley Longman, Inc., Reading, MA (1994) ISBN 0-201-53393-6.

[NEIDER93] Neider, Jackie, Tom Davis, and Mason Woo, OpenGL Programming Guide, Addison Wesley Inc., Reading, MA (1993) ISBN 0-201-63274-8.

[PERSON68] Person, Russell V., Essentials of Mathematics, 2nd Edition, John Wiley & Sons, Inc., New York, NY (1968) ISBN 0-132-84191-6.

[PLAUGER92] Plauger, P.J., The Standard C Library, Prentice-Hall, Englewood Cliffs, NJ (1992) ISBN 0-13-131509-9.

[SEDGEWICK88] Sedgewick, Robert, Algorithms, 2nd Edition, Addison Wesley Longman, Inc., Reading, MA (1988) ISBN 0-201-06673-4.

[SHAMPINE97] Shampine, L.E, R.C. Alien, Jr., and S. Pruess, Fundamentals of Numerical Computing, John Wiley & Sons, Inc., New York, NY (1997) ISBN 0-471-16363-5.

[STROUSTRUP94] Stroustrup, Bjarne, The Design and Evolution of C++, Addison Wesley Longman, Inc., Reading, MA (1994) ISBN 0-201-54330-3.

[STROUSTRUP97] Stroustrup, Bjarne, The C++ Programming Language, 3rd Edition, Addison Wesley Longman, Inc., Reading, MA (1997) ISBN 0-201-88954-4.

[UPSTILL90] Upstill, Steve, The RenderMan Companion, Addison Wesley Longman, Inc., Reading, MA (1990) ISBN 0-201-50868-0.

[WERNECKE94] Wernecke, Josie, The Inventor Mentor, Addison Wesley Longman, Inc., Reading, MA (1994) ISBN 0-201-62495-8.

[YOUNG95] Young, Douglas A., Object-Oriented Programming with C++ and OSF/ Motif, 2nd Edition, Prentice-Hall, Englewood Cliffs, NJ (1995) ISBN 0-132-09255-7.

Часть I

Краткий обзор языка C++

Программы, которые мы пишем, имеют два основных аспекта:

набор алгоритмов;

набор *данных*, которыми оперируют.

Эти два аспекта оставались неизменными за всю недолгую историю программирования, зато отношения между ними *(парадигма программирования)* менялись. .

В *процедурной* парадигме программирования задача непосредственно моделируется набором алгоритмов. Возьмем, к примеру, систему выдачи книг в библиотеке. В ней реализуются две главные процедуры: процедура выдачи книг и процедура приема книг. Данные хранятся отдельно и передаются этим процедурам как параметры. К наиболее известным процедурным языкам программирования относятся FORTRAN, C и Pascal. C++ также поддерживает процедурное программирование. Отдельные процедуры носят в этом языке название функций. В части III рассматривается поддержка, предоставляемая в C++ процедурной парадигме программирования: функции, шаблоны функций, обобщенные алгоритмы.

В 70-е годы процедурную парадигму стала вытеснять парадигма *абстрактных типов данных* (теперь чаще называемая объектным подходом). В рамках этой парадигмы задача моделируется набором абстракций данных. В C++ эти абстракции получили название *классов*. Наша библиотечная система могла бы быть представлена как взаимоотношения объектов различных классов, представляющих книги, читателей, даты возврата и т.п. Алгоритмы, реализуемые каждым классом, называются *открытым интерфейсом класса*. Данные “скрыты” внутри объектов класса. Парадигму абстрактных типов данных поддерживают такие языки, как CLU, Ada и Modula-2. В части IV обсуждаются вопросы поддержки этой парадигмы языком C++.

Объектно-ориентированное программирование расширяет парадигму абстрактных типов данных механизмом *наследования* (повторного использования существующих объектов) и *динамического связывания* (повторного использования существующих интерфейсов). Вводятся отношения тип-подтип. Книга, видеокассета, компакт-диск – все они хранятся в библиотеке, и поэтому могут быть названы подтипами (или подклассами) одного родительского типа, представляющего то, что может храниться в библиотеке. Хотя каждый из классов способен реализовывать свой собственный алгоритм выдачи и возврата, открытый интерфейс для них одинаков. Три наиболее известных языка, поддерживающие объектно-ориентированный подход, – это Simula, Smalltalk и Java. В части V рассматриваются вопросы поддержки парадигмы объектно-ориентированного программирования в C++.

Хотя мы и считаем C++ в основном объектно-ориентированным языком, он поддерживает и процедурную, и объектную парадигму. Преимущество такого подхода в том, что для каждого конкретного случая можно выбрать наилучшее решение. Однако есть и обратная сторона медали: C++ является достаточно громоздким и сложным языком.

В части I мы “пробежимся” по всем основным аспектам C++. Одна из причин такого краткого обзора – желание дать читателю представление об основных возможностях языка, чтобы затем приводить достаточно содержательные примеры. Скажем, мы не будем рассматривать в деталях понятие класса вплоть до главы 13, однако без упоминания о нем наши примеры оказались бы неинтересными и надуманными.

Другая причина такого поверхностного, но широкого обзора – эстетическая. Если вы еще не оценили красоту и сложность сонаты Бетховена или живость регтайма Джоплина, вам будет безумно скучно разбираться в отдельных деталях вроде диезов, бемолей, октав и аккордов. Однако, не овладев ими, вы не научитесь музыке. Во многом это справедливо и для программирования. Разбираться в путанице приоритетов операций или правил приведения типов скучно, но совершенно необходимо для овладения C++.

В главе 1 представлены базовые элементы языка: встроенные типы данных, переменные, выражения, инструкции (statements) и функции. Мы увидим минимальную законченную C++ программу, обсудим вопросы компиляции, коснемся препроцессора и поддержки ввода/вывода.

В главе 2 мы реализуем абстракцию массива – процедурно, объектно, и объектно-ориентированно. Мы сравним нашу реализацию с реализацией, предоставляемой стандартной библиотекой C++, и познакомимся с набором обобщенных алгоритмов стандартной библиотеки. Мы коснемся и таких вещей, как шаблоны, исключения и пространства имен. Фактически, мы представим все особенности языка C++, хотя обсуждение деталей отложим до следующих глав.

Возможно, некоторые читатели сочтут главу 2 трудной для понимания. Материал представляется без подробного объяснения, даются ссылки на последующие разделы. Мы рекомендуем таким читателям не углубляться в эту главу, пропустить ее вовсе или прочитать по диагонали. В главе 3 материал излагается в более традиционной манере. После этого можно будет вернуться к главе 2.

1. Начинаем

В этой главе представлены основные элементы языка: встроенные типы данных, определения именованных объектов, выражений и операторов, определение и использование именованных функций. Мы посмотрим на минимальную законченную C++ программу, вкратце коснемся процесса компиляции этой программы, узнаем, что такое препроцессор, и бросим самый первый взгляд на поддержку ввода и вывода. Мы увидим также ряд простых, но законченных C++ программ.

1.1. Решение задачи

Программы обычно пишутся для того, чтобы решить какую-то конкретную задачу. Например, книжный магазин ведет запись проданных книг. Регистрируется название книги и издательство, причем запись идет в том порядке, в каком книги продаются. Каждые две недели владелец магазина вручную подсчитывает количество проданных книг с одинаковым названием и количество проданных книг от каждого издателя. Этот список сортируется по издателям и используется для составления последующего заказа книг. Нас попросили написать программу для автоматизации этой деятельности.

Один из методов решения большой задачи состоит в разбиении ее на ряд задач поменьше. В идеале, с маленькими задачами легче справиться, а вместе они помогают одолеть большую. Если подзадачи все еще слишком сложны, мы, в свою очередь, разобьем их на еще меньшие, пока каждая из подзадач не будет решена. Такую стратегию называют *пошаговой детализацией* или принципом *“разделяй и властвуй”*. Задача книжного магазина делится на четыре подзадачи:

Прочитать файл с записями о продажах.

Подсчитать количество продаж по названиям и по издателям.

Отсортировать записи по издателям.

Вывести результаты.

Решения для подзадач 1, 2 и 4 известны, их не нужно делить на более мелкие подзадачи. А вот третья подзадача все еще слишком сложна. Будем дробить ее дальше.

3a. Отсортировать записи по издателям.

3b. Для каждого издателя отсортировать записи по названиям.

3c. Сравнить соседние записи в группе каждого издателя. Для каждой одинаковой пары увеличить счетчик для первой записи и удалить вторую.

Эти подзадачи решаются легко. Теперь мы знаем, как решить исходную, большую задачу. Более того, мы видим, что первоначальный список подзадач был не совсем правильным. Правильная последовательность действий такова:

Прочитать файл с записями о продажах.

Отсортировать этот файл: сначала по издателям, внутри каждого издателя – по названиям.

Удалить повторяющиеся названия, наращивая счетчик.

Вывести результат в новый файл.

Результирующая последовательность действий называется *алгоритмом*. Следующий шаг – перевести наш алгоритм на некоторый язык программирования, в нашем случае – на C++.

1.2. Программа на языке C++

В C++ действие называется *выражением*, *а* выражение, заканчивающееся точкой с запятой, – *инструкцией*. Инструкция – это атомарная часть C++ программы, которой в программе на C++ соответствует предложение естественного языка. Вот примеры

```
int book_count = 0;
book_count = books_on_shelf + books_on_order;
```

инструкций C++:

```
cout << "значение переменной book_count: " << book_count;
```

Первая из приведенных инструкций является инструкцией *объявления*. `book_count` можно назвать *идентификатором*, *символической переменной* (или просто *переменной*) или *объектом*. Переменной соответствует область в памяти компьютера, соотнесенная с определенным именем (в данном случае `book_count`), в которой хранится значение типа (в нашем случае целого). `0` – это *константа*. Переменная `book_count` *инициализирована* значением `0`.

Вторая инструкция – *присваивание*. Она помещает в область памяти, отведенную переменной `book_count`, результат сложения двух других переменных – `books_on_shelf` и `books_on_order`. Предполагается, что эти две целочисленные переменные определены где-то ранее в программе и им присвоены некоторые значения.

Третья инструкция является инструкцией *вывода*. cout – это выходной поток, направленный на терминал, << – оператор вывода. Эта инструкция выводит в cout – то есть на терминал – сначала символьную константу, заключенную в двойные кавычки ("значение переменной book_count: "), затем значение, содержащееся в области памяти, отведенном под переменную book_count. В результате выполнения данной инструкции мы получим на терминале сообщение:

```
значение переменной book_count: 11273
```

если значение book_count равно 11273 в данной точке выполнения программы.

Инструкции часто объединяются в именованные группы, называемые *функциями*. Так, группа инструкций, необходимых для чтения исходного файла, объединена в функцию readIn(). Аналогичным образом инструкции для выполнения оставшихся подзадач сгруппированы в функции sort(), compact() и print().

В каждой C++ программе должна быть ровно одна функция с именем main(). Вот как

```
int main()
{
  readIn();
  sort();
  compact();
  print();

  return 0;
}
```

может выглядеть эта функция для нашего алгоритма:

Исполнение программы начинается с выполнения первой инструкции функции main(), в нашем случае – вызовом функции readIn(). Затем одна за другой исполняются все дальнейшие инструкции, и, выполнив последнюю инструкцию функции main(), программа заканчивает работу.

Функция состоит их четырех частей: типа возвращаемого значения, имени, списка параметров и тела функции. Первые три части составляют *прототип функции*.

Список параметров заключается в круглые скобки и может содержать ноль или более параметров, разделенных запятыми. Тело функции содержит последовательность исполняемых инструкций и ограничено фигурными скобками.

В нашем примере тело функции main() содержит *вызовы* функций readIn(), sort(), compact() и print(). Последней выполняется инструкция

```
return 0;
```

Инструкция return обеспечивает механизм завершения работы функции. Если оператор return сопровождается некоторым значением (в данном примере 0), это значение становится *возвращаемым значением* функции. В нашем примере возвращаемое значение 0 говорит об успешном выполнении функции main(). (Стандарт C++ предусматривает, что функция main() возвращает 0 по умолчанию, если оператор return не использован явно.)

Давайте закончим нашу программу, чтобы ее можно было откомпилировать и выполнить. Во-первых, мы должны определить функции `readIn()`, `sort()`, `compact()` и `print()`.

```
void readIn() { cout << "readIn()\n"; }
void sort() { cout << "sort()\n"; }
```

Для начала вполне подойдут заглушки:

```
void compact() { cout << "compact()\n"; }
void print() { cout << "print ()\n"; }
```

Тип `void` используется, чтобы обозначить функцию, которая не возвращает никакого значения. Наши заглушки не производят никаких полезных действий, они только выводят на терминал сообщения о том, что были вызваны. Впоследствии мы заменим их на реальные функции, выполняющие нужную нам работу.

Пошаговый метод написания программ позволяет справляться с неизбежными ошибками. Попытаться заставить работать сразу всю программу – слишком сложное занятие.

Имя файла с текстом программы, или исходного файла, как правило, состоит из двух частей: собственно имени (например, `bookstore`) и расширения, записываемого после точки. Расширение, в соответствии с принятыми соглашениями, служит для определения назначения файла. Файл `bookstore.h` является *заголовочным файлом* для C или C++ программы. (Необходимо отметить, что стандартные заголовочные файлы C++ являются исключением из правила: у них нет расширения.)

Файл bookstore.c является исходным файлом для нашей C программы. В операционной системе UNIX, где строчные и прописные буквы в именах файлов различаются, расширение .C обозначает исходный текст C++ программы, и в файле bookstore.C располагается исходный текст C++.

В других операционных системах, в частности в DOS, где строчные и прописные буквы не различаются, разные реализации могут использовать разные соглашения для обозначения исходных файлов C++. Чаще всего употребляются расширения `.cpp` и `.cxx`: `bookstore.cpp`, `bookstore.cxx`.

Заголовочные файлы C++ программ также могут иметь разные расширения в разных реализациях (и это одна из причин того, что стандартные заголовочные файлы C++ не имеют расширения). Расширения, используемые в конкретной реализации компилятора C++, указаны в поставляемой вместе с ним документации.

Итак, создадим текст законченной C++ программы (используя любой текстовый редактор):

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

void readIn() { cout << "readIn()\n"; }
void sort() { cout << "sort()\n"; }
void compact() { cout << "compact()\n"; }
void print() { cout << "print ()\n"; }
int main()
{
  readIn();
  sort();
  compact();
  print();

  return 0;
}
```

Здесь `iostream` – стандартный заголовочный файл библиотеки ввода/вывода (обратите внимание: у него нет расширения). Эта библиотека содержит информацию о потоке `cout`, используемом в нашей программе. `#include` является *директивой препроцессора*, заставляющей включить в нашу программу текст из заголовочного файла `iostream`. (Директивы препроцессора рассматриваются в разделе 1.3.)

Непосредственно за директивой препроцессора

```
#include <iostream>
```

следует инструкция

```
using namespace std;
```

Эта инструкция называется директивой `using`. Имена, используемые в стандартной библиотеке C++ (такие, как `cout`), объявлены в пространстве имен `std` и невидимы в нашей программе до тех пор, пока мы явно не сделаем их видимыми, для чего и применяется данная директива. (Подробнее о пространстве имен говорится в разделах 2.7 и 8.5.)1

После того как исходный текст программы помещен в файл, скажем `prog1.C`, мы должны откомпилировать его. В UNIX для этого выполняется следующая команда:

```
$ CC prog1.C
```

Здесь $ представляет собой приглашение командной строки. CC – команда вызова компилятора C++, принятая в большинстве UNIX-систем. Команды вызова компилятора могут быть разными в разных системах.

Одной из задач, выполняемых компилятором в процессе обработки исходного файла, является проверка правильности программы. Компилятор не может обнаружить

---

1 Во время написания этой книги не все компиляторы C++ поддерживали пространства имен. Если ваш компилятор таков, откажитесь от данной директивы. Большинство программ, приводимых нами, используют компиляторы, не поддерживающие пространство имен, поэтому директива `using` в них отсутствует.

смысловые ошибки, однако он может найти формальные ошибки в тексте программы. Существует два типа формальных ошибок:

синтаксические ошибки. Программист может допустить “грамматические”, с точки

```
int main( { // ошибка - пропущена ')'
  readIn(): // ошибка - недопустимый символ ':'
  sort();
  compact();
  print();

  return 0  // ошибка - пропущен символ ';'
}
```

зрения языка C++, ошибки. Например:

ошибки типизации. С каждой переменной и константой в C++ сопоставлен некоторый тип. Например, число 10 – целого типа. Строка `"hello"`, заключенная в двойные кавычки, имеет символьный тип. Если функция ожидает получить в качестве параметра целое значение, а получает символьную строку, компилятор рассматривает это как ошибку типизации.

Сообщение об ошибке содержит номер строки и краткое описание. Полезно просматривать список ошибок, начиная с первой, потому что одна-единственная ошибка может вызвать цепную реакцию, появление “наведенных” ошибок. Исправление этой единственной ошибки приведет и к исчезновению остальных. После исправления синтаксических ошибок программу нужно перекомпилировать.

После проверки на правильность компилятор переводит исходный текст в объектный код, который может быть понят и исполнен компьютером. Эту фазу работы компилятора называют *генерацией кода*.

В результате успешной компиляции образуется выполняемый файл. Если запустить выполняемый файл, полученный в результате компиляции нашей программы, на

```
readIn()
sort()
compact()
```

терминале появится следующий текст:

```
print()
```

В C++ набор основных типов данных – это целый и вещественный числовые типы, символьный тип и логический, или булевский. Каждый тип обозначается своим ключевым словом. Любой объект программы ассоциируется с некоторым типом.

```
int    age = 10;
double price = 19.99;
char   delimiter = ' ';
```

Например:

```
bool   found = false;
```

Здесь определены четыре объекта: age, price, delimiter, found, имеющие соответственно типы целый, вещественный с двойной точностью, символьный и логический. Каждый объект инициализирован константой – целым числом 10, вещественным числом 19.99, символом пробела и логическим значением false.

Между основными типами данных может осуществляться неявное *преобразование типов*. Если переменной age, имеющей тип int, присвоить константу типа double, например:

```
age = 33.333;
```

то значением переменной age станет целое число 33. (Стандартные преобразования типов, а также общие проблемы преобразования типов рассматриваются в разделе 4.14.)

Стандартная библиотека C++ расширяет базовый набор типов, добавляя к ним такие

```
// заголовочный файл с определением типа string
#include <string>
string current_chapter = "Начинаем";

// заголовочный файл с определением типа vector
#include <vector>
```

типы, как строка, комплексное число, вектор, список. Примеры:

```
vector<string> chapter_titles(20);
```

Здесь current_chapter – объект типа string, инициализированный константой "Начинаем". Переменная chapter_titles – вектор из 20 элементов строкового типа. Несколько необычный синтаксис выражения

```
vector<string>
```

сообщает компилятору о необходимости создать вектор, содержащий объекты типа string. Для того чтобы определить вектор из 20 целых значений, необходимо написать:

```
vector<int> ivec(20);
```

Никакой язык, никакие стандартные библиотеки не способны обеспечить нас всеми типами данных, которые могут потребоваться. Взамен современные языки программирования предоставляют механизм создания новых типов данных. В C++ для этого служит механизм классов. Все расширенные типы данных из стандартной библиотеки C++, такие как строка, комплексное число, вектор, список, являются классами, написанными на C++. Классами являются и объекты из библиотеки ввода/вывода.

Механизм классов – одна из самых главных особенностей языка C++, и в главе 2 мы рассмотрим его очень подробно.

1.2.1. Порядок выполнения инструкций

По умолчанию инструкции программы выполняются одна за другой, последовательно. В программе

```
int main()
{
  readIn();
  sort();
  compact();
  print();

  return 0;
}
```

первой будет выполнена инструкция `readIn()`, за ней `sort()`, `compact()` и наконец `print()`.

Однако представим себе ситуацию, когда количество продаж невелико: оно равно 1 или даже `0`. Вряд ли стоит вызывать функции `sort()` и `compact()` для такого случая. Но вывести результат все-таки нужно, поэтому функцию `print()` следует вызывать в любом случае. Для этого случая мы можем использовать *условную инструкцию* `if`. Нам придется переписать функцию `readIn()` так, чтобы она возвращала количество прочитанных

```
// readIn() возвращает количество прочитанных записей
// возвращаемое значение имеет тип int
int readIn() { ... }

// ...

int main()
{
  int count = readIn();

  // если количество записей больше 1,
  // то вызвать sort() и compact()

  if ( count > 1 ) {
    sort();
    compact();
  }

  if ( count == 0 )
    cout << "Продаж не было\n";
  else
    print();

  return 0;
}
```

записей:

Первая инструкция `if` обеспечивает условное выполнение блока программы: функции `sort()` и `compact()` вызываются только в том случае, если `count` больше `1`. Согласно второй инструкции `if` на терминал выводится сообщение “Продаж не было”, если условие истинно, т.е. значение `count` равно `0`. Если же это условие ложно, производится вызов функции `print()`. (Детальное описание инструкции `if` приводится в разделе 5.3.)

Другим распространенным способом непоследовательного выполнения программы является итерация, или инструкция *цикла*. Такая инструкция предписывает повторять

блок программы до тех пор, пока некоторое условие не изменится с `true` на `false`.

```
int main()
{
  int iterations = 0;
  bool continue_loop = true;
  while ( continue_loop != false )
  {
    iterations++;

    cout << "Цикл был выполнен " << iterations << "раз\n";

    if ( iterations == 5 )
      continue_loop = false;
  }
  return 0;
```

Например:

```
}
```

В этом надуманном примере цикл `while` выполняется пять раз, до тех пор пока переменная `iterations` не получит значение `5` и переменная `continue_loop` не станет равной `false`. Инструкция

```
iterations++;
```

увеличивает значение переменной `iterations` на единицу. (Инструкции цикла детально рассматриваются в главе 5.)

1.3. Директивы препроцессора

Заголовочные файлы включаются в текст программы с помощью *директивы препроцессора* `#include`. Директивы препроцессора начинаются со знака “диез” (#), который должен быть самым первым символом строки. Программа, которая обрабатывает эти директивы, называется *препроцессором* (в современных компиляторах препроцессор обычно является частью самого компилятора).

Директива `#include` включает в программу содержимое указанного файла. Имя файла

```
#include <some_file.h>
```

может быть указано двумя способами:

```
#include "my_file.h"
```

Если имя файла заключено в угловые скобки (<>), считается, что нам нужен некий стандартный заголовочный файл, и компилятор ищет этот файл в предопределенных местах. (Способ определения этих мест сильно различается для разных платформ и реализаций.) Двойные кавычки означают, что заголовочный файл – пользовательский, и его поиск начинается с того каталога, где находится исходный текст программы.

Заголовочный файл также может содержать директивы `#include`. Поэтому иногда трудно понять, какие же конкретно заголовочные файлы включены в данный исходный

текст, и некоторые заголовочные файлы могут оказаться включенными несколько раз. Избежать этого позволяют *условные директивы препроцессора*. Рассмотрим пример:

```
#define BOOKSTORE_H
  /* содержимое файла bookstore.h */

#ifndef BOOKSTORE_H
#endif
```

Условная директива #ifndef проверяет, не было ли значение BOOKSTORE_H определено ранее. (BOOKSTORE_H – это константа препроцессора; такие константы принято писать заглавными буквами.) Препроцессор обрабатывает следующие строки вплоть до директивы #endif. В противном случае он пропускает строки от #ifndef до # endif.

Директива

```
#define BOOKSTORE_H
```

определяет константу препроцессора BOOKSTORE_H. Поместив эту директиву непосредственно после директивы #ifndef, мы можем гарантировать, что содержательная часть заголовочного файла bookstore.h будет включена в исходный текст только один раз, сколько бы раз ни включался в текст сам этот файл.

Другим распространенным примером применения условных директив препроцессора

```
int main()
{
#ifdef DEBUG
  cout << "Начало выполнения main()\n";
#endif

  string word;
  vector<string> text;

  while ( cin >> word )
  {
#ifdef DEBUG
    cout << "Прочитано слово: " << word << "\n";
#endif
    text.push_back(word);
  }
  // ...
```

является включение в текст программы отладочной информации. Например:

```
}
```

Если константа DEBUG не определена, результирующий текст программы будет выглядеть

```
int main()
```

так:

```
{

  string word;
```

```
  vector<string> text;

  while ( cin >> word )
  {
    text.push_back(word);
  }
  // ...
}


int main()
```

В противном случае мы получим:

```
{
  cout << "Начало выполнения main()\n";

  string word;
  vector<string> text;

  while ( cin >> word )
  {
    cout << "Прочитано слово: " << word << "\n";
    text.push_back(word);
  }
  // ...
}
```

Константа препроцессора может быть определена в командной строке при вызове компилятора с помощью опции `-D` (в различных реализациях эта опция может называться по-разному). Для UNIX-систем вызов компилятора с определением препроцессорной константы `DEBUG` выглядит следующим образом:

```
$ CC -DDEBUG main.C
```

Есть константы, которые автоматически определяются компилятором. Например, мы можем узнать, компилируем ли мы C++ или C программу. Для C++ программы автоматически определяется константа `__cplusplus` (два подчеркивания). Для стандартного C определяется `__STDC__`. Естественно, обе константы не могут быть

```
#idfef __cplusplus
  // компиляция C++ программы
  extern "C";
  // extern "C" объясняется в главе 7
#endif
```

определены одновременно. Пример:

```
int main(int,int);
```

Другими полезными предопределенными константами (в данном случае лучше сказать переменными) препроцессора являются `__LINE__` и `__FILE__`. Переменная `__LINE__` содержит номер текущей компилируемой строки, а `__FILE__` – имя компилируемого файла. Вот пример их использования:

```
if ( element_count == 0 )

  cerr << "Ошибка. Файл: " << __FILE__
       << " Строка: " << __LINE__
       << "element_count не может быть 0";
```

Две константы __DATE__ и __TIME__ содержат дату и время компиляции.

Стандартная библиотека C предоставляет полезный макрос assert(), который проверяет некоторое условие и в случае, если оно не выполняется, выдает диагностическое сообщение и аварийно завершает программу. Мы будем часто пользоваться этим полезным макросом в последующих примерах программ. Для его применения следует включить в программу директиву

```
#include <assert.h>
```

assert.h – это заголовочный файл стандартной библиотеки C. Программа на C++ может ссылаться на заголовочный файл как по его имени, принятому в C, так и по имени, принятому в C++. В стандартной библиотеке C++ этот файл носит имя cassert. Имя заголовочного файла в библиотеке C++ отличается от имени соответствующего файла для C отсутствием расширения .h и подставленной спереди буквой c (выше уже упоминалось, что в заголовочных файлах для C++ расширения не употребляются, поскольку они могут зависеть от реализации).

Эффект от использования директивы препроцессора #include зависит от типа заголовочного файла. Инструкция

```
#include <cassert>
```

включает в текст программы содержимое файла cassert. Но поскольку все имена, используемые в стандартной библиотеке C++, определены в пространстве std, имя assert() будет невидимо до тех пор, пока мы явно не сделаем его видимым с помощью следующей using-директивы:

using namespace std;

Если же мы включаем в программу заголовочный файл для библиотеки C

```
#include <assert.h>
```

то надобность в using-директиве отпадает: имя assert() будет видно и так2. (Пространства имен используются разработчиками библиотек для предотвращения засорения глобального пространства имен. В разделе 8.5 эта тема рассматривается более подробно.)

1.4. Немного о комментариях

Комментарии помогают человеку читать текст программы; писать их грамотно считается правилом хорошего тона. Комментарии могут характеризовать используемый алгоритм,

---

2 Как было сказано ранее, не все компиляторы поддерживают пространства имен, поэтому эта разница проявляется только для последних версий компиляторов.

пояснять назначение тех или иных переменных, разъяснять непонятные места. При компиляции комментарии выкидываются из текста программы поэтому размер получающегося исполняемого модуля не увеличивается.

В C++ есть два типа комментариев. Один – такой же, как и в C, использующий символы /* для обозначения начала и */ для обозначения конца комментария. Между этими парами символов может находиться любой текст, занимающий одну или несколько строк:

```
/*
 * Это первое знакомство с определением класса в C++.
 * Классы используются как в объектном, так и в
 * объектно-ориентированном программировании. Реализация
 * класса Screen представлена в главе 13.
*/
class Screen {
    /* Это называется телом класса */
public:
    void home();    /* переместить курсор в позицию 0,0 */
    void refresh ();/* перерисовать экран               */
private:
    /* Классы поддерживают "сокрытие информации"    */
    /* Сокрытие информации ограничивает доступ из   */
    /* программы к внутреннему представлению класса */
    /* (его данным). Для этого используется метка   */
    /* "private:"                                   */
    int height, width;
```

вся последовательность между /* и */ считается комментарием. Например:

```
}
```

Слишком большое число комментариев, перемежающихся с кодом программы, может ухудшить читаемость текста. Например, объявления переменных width и height в данном тексте окружены комментариями и почти не заметны. Рекомендуется писать развернутое объяснение перед блоком текста. Как и любая программная документация, комментарии должны обновляться в процессе модификации кода. Увы, нередко случается, что они относятся к устаревшей версии.

Комментарии в стиле C не могут быть вложенными. Попробуйте откомпилировать нижеследующую программу в своей системе. Большинство компиляторов посчитают ее

```
#include <iostream>

/* комментарии /* */ не могут быть вложенными.
 * Строку "не вкладываются" компилятор рассматривает,
 * как часть программы. Это же относится к данной и следующей строкам
 */

int main() {
    cout << "Здравствуй, мир\n";
```

ошибочной:

```
}
```

Один из способов решить проблему вложенных комментариев – поставить пробел между звездочкой и косой чертой:

```
/* * /
```

Последовательность символов */ считается концом комментария только в том случае, если между ними нет пробела.

Второй тип комментариев – однострочный. Он начинается последовательностью символов // и ограничен концом строки. Часть строки вправо от двух косых черт игнорируется компилятором. Вот пример нашего класса Screen с использованием двух

```
/*
 * Первое знакомство с определением класса в C++.
 * Классы используются как в объектном, так и в
 * объектно-ориентированном программировании. Реализация
 * класса Screen представлена в главе 13.
 */

class Screen {
    // Это называется телом класса
public:
    void home();     // переместить курсор в позицию 0,0
    void refresh (); // перерисовать экран
private:
    /* Классы поддерживают "сокрытие информации".    */
    /* Сокрытие информации ограничивает доступ из   */
    /* программы к внутреннему представлению класса */
    /* (его данным). Для этого используется метка   */
    /* "private:"                                    */
    int height, width;
```

строчных комментариев:

```
}
```

Обычно в программе употребляют сразу оба типа комментариев. Строчные комментарии удобны для кратких пояснений – в одну или полстроки, а комментарии, ограниченные /* и */, лучше подходят для развернутых многострочных пояснений.

1.5. Первый взгляд на ввод/вывод

Частью стандартной библиотеки C++ является библиотека iostream, которая реализована как иерархия классов и обеспечивает базовые возможности ввода/вывода.

Ввод с терминала, называемый *стандартным вводом*, “привязан” к предопределенному объекту cin. Вывод на терминал, или *стандартный вывод*, привязан к объекту cout. Третий предопределенный объект, cerr, представляет собой *стандартный вывод для ошибок*. Обычно он используется для вывода сообщений об ошибках и предупреждений.

Для использования библиотеки ввода/вывода необходимо включить соответствующий заголовочный файл:

```
#include <iostream>
```

Чтобы значение поступило в стандартный вывод или в стандартный вывод для ошибок используется оператор <<:

```
int v1, v2;
// ...
cout << "сумма v1 и v2 = ";
cout << v1 + v2;
```

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

Последовательность "\n" представляет собой символ перехода на новую строку. Вместо "\n" мы можем использовать предопределенный *манипулятор* endl.

```
cout << endl;
```

Манипулятор endl не просто выводит данные (символ перехода на новую строку), но и производит сброс буфера вывода. (Предопределенные манипуляторы рассматриваются в главе 20.)

Операторы вывода можно сцеплять. Так, три строки в предыдущем примере заменяются одной:

```
cout << "сумма v1 и v2 = " << v1 + v2 << "\n";
```

```
string file_name;
// ...
cout << "Введите имя файла: ";
```

Для чтения значения из стандартного ввода применяется оператор ввода (>>):

```
cin >> file_name;
```

```
string ifile, ofile;
```

Операторы ввода, как и операторы вывода, можно сцеплять:

```
// ...
cout << "Введите имя входного и выходного файлов: ";
cin >> ifile >> ofile;
```

Каким образом ввести заранее неизвестное число значений? Мы вернемся к этому

```
string word;
```

вопросу в конце раздела 2.2, а пока скажем, что последовательность инструкций

```
while ( cin >> word )
  // ...
```

считывает по одному слову из стандартного ввода до тех пор, пока не считаны все слова. Выражение

```
( cin >> word )
```

возвращает false, когда достигнут конец файла. (Подробнее об этом – в главе 20.) Вот пример простой законченной программы, считывающей по одному слову из cin и

```
#include <iostream>
#include <string>

int main ()
{
  string word;

  while ( cin >> word )
    cout << "Прочитано слово: " << word << "\n";

  cout << "Все слова прочитаны!";
```

выводящей их в cout:

```
}
```

Вот первое предложение из произведения Джеймса Джойса “Пробуждение Финнегана”:

```
riverrun, past Eve and Adam's
```

Если запустить приведенную выше программу и набрать с клавиатуры данное предложение, мы увидим на экране терминала следующее:

```
Прочитано слово: riverrun,
Прочитано слово: past
Прочитано слово: Eve,
Прочитано слово: and
Прочитано слово: Adam's
```

```
Все слова прочитаны!
```

(В главе 6 мы рассмотрим вопрос о том, как убрать знаки препинания из вводимых слов.)

1.5.1. Файловый ввод/вывод

Библиотека iostream поддерживает и файловый ввод/вывод. Все операции, применимые в стандартному вводу и выводу, могут быть также применены к файлам. Чтобы использовать файл для ввода или вывода, мы должны включить еще один заголовочный файл:

```
#include <fstream>
```

Перед тем как открыть файл для вывода, необходимо объявить объект типа ofstream:

```
ofstream outfile("name-of-file");
```

Проверить, удалось ли нам открыть файл, можно следующим образом:

```
if ( ! outfile ) // false, если файл не открыт
   cerr << "Ошибка открытия файла.\n"
```

```
ifstream infile("name-of-file");
```

Так же открывается файл и для ввода, только он имеет тип ifstream:

```
if ( ! infile ) // false, если файл не открыт
   cerr << "Ошибка открытия файла.\n"
```

Ниже приводится текст простой программы, которая читает файл с именем in_file и выводит все прочитанные из этого файла слова, разделяя их пробелом, в другой файл,

```
#include <iostream>
#include <fstream>
#include <string>

int main()
{
  ifstream infile("in_file");
  ofstream outfile("out_file");

  if ( ! infile ) {
    cerr << "Ошибка открытия входного файла.\n"
    return -1;
  }
  if ( ! outfile ) {
    cerr << "Ошибка открытия выходного файла.\n"
    return -2;
  }
```

названный out_file.

```
  string word;
  while ( infile >> word )
    outfile << word << ' ';

  return 0;
}
```

В главе 20 библиотека ввода/вывода будет рассмотрена подробно. А в следующих разделах мы увидим, как можно создавать новые типы данных, используя механизм классов и шаблонов.

2. Краткий обзор C++

Эту главу мы начнем с рассмотрения встроенного в язык C++ типа данных “массив”. Массив – это набор данных одного типа, например массив целых чисел или массив строк. Мы рассмотрим недостатки, присущие встроенному массиву, и напишем для его представления свой класс Array, где попытаемся избавиться от этих недостатков. Затем мы построим целую иерархию подклассов, основываясь на нашем базовом классе Array. В конце концов мы сравним наш класс Array с классом vector из стандартной библиотеки C++, реализующим аналогичную функциональность. В процессе создания этих классов мы коснемся таких свойств C++, как шаблоны, пространства имен и обработка ошибок.

2.1. Встроенный тип данных “массив”

Как было показано в главе 1, C++ предоставляет встроенную поддержку для основных

```
// объявление целого объекта ival
// ival инициализируется значением 1024
int ival = 1024;

// объявление вещественного объекта двойной точности dval
// dval инициализируется значением 3.14159
double dval = 3.14159;

// объявление вещественного объекта одинарной точности fval
// fval инициализируется значением 3.14159
```

типов данных – целых и вещественных чисел, логических значений и символов:

```
float fval = 3.14159;
```

К числовым типам данных могут применяться встроенные арифметические и логические

```
int ival2 = ival1 + 4096;   // сложение
```

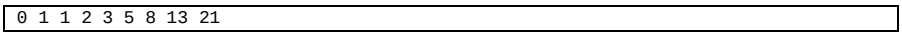
операции: объекты числового типа можно складывать, вычитать, умножать, делить и т.д.

```
int ival3 = ival2 - ival;   // вычитание

dval = fval * ival;         // умножение
ival = ival3 / 2;           // деление

bool result = ival2 == ival3;   // сравнение на равенство
result = ival2 + ival != ival3; // сравнение на неравенство
result = fval + ival2 < dval;   // сравнение на меньше
result = ival > ival2;          // сравнение на больше
```

В дополнение к встроенным типам стандартная библиотека C++ предоставляет поддержку для расширенного набора типов, таких, как строка и комплексное число. (Мы отложим рассмотрение класса vector из стандартной библиотеки до раздела 2.7.)

Промежуточное положение между встроенными типами данных и типами данных из стандартной библиотеки занимают составные типы – массивы и указатели. (Указатели рассмотрены в разделе 2.2.)

Массив – это упорядоченный набор элементов одного типа. Например, последовательность



представляет собой первые 9 элементов последовательности Фибоначчи. (Выбрав начальные два числа, вычисляем каждый из следующих элементов как сумму двух предыдущих.)

Для того чтобы объявить массив и проинициализировать его данными элементами, мы должны написать следующую инструкцию C++:

```
int fibon[9] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
```

Здесь `fibon` – это имя массива. Элементы массива имеют тип `int`, *размер* (длина) массива равна 9. Значение первого элемента – `0`, последнего – `21`. Для работы с массивом мы *индексируем* (нумеруем) его элементы, а доступ к ним осуществляется с помощью операции *взятия индекса*. Казалось бы, для обращения к первому элементу массива естественно написать:

```
int first_elem = fibon[1];
```

Однако это не совсем правильно: в C++ (как и в C) индексация массивов начинается с 0, поэтому элемент с индексом 1 на самом деле является *вторым* элементом массива, а индекс первого равен 0.Таким образом, чтобы обратиться к последнему элементу

```
fibon[0]; // первый элемент
fibon[1]; // второй элемент
...
fibon[8]; // последний элемент
```

массива, мы должны вычесть единицу из размера массива:

```
fibon[9]; // ... ошибка
```

Девять элементов массива `fibon` имеют индексы от 0 до 8. Употребление вместо этого индексов 1-9 является одной из самых распространенных ошибок начинающих программистов на C++.

Для перебора элементов массива обычно употребляют инструкцию цикла. Вот пример программы, которая инициализирует массив из десяти элементов числами от 0 до 9 и

```
int main()
```

затем печатает их в обратном порядке:

```
{
  int ia[10];
  int index;
```

```
  for (index=0; index<10; ++index)
    // ia[0] = 0, ia[1] = 1 и т.д.
    ia[index] = index;

  for (index=9; index>=0; --index)
    cout << ia[index] << " ";

  cout << endl;
}
```

Оба цикла выполняются по 10 раз. Все управление циклом `for` осуществляется инструкциями в круглых скобках за ключевым словом `for`. Первая присваивает начальное значение переменной `index`. Это производится один раз перед началом цикла:

```
index = 0;
```

Вторая инструкция:

```
index < 10;
```

представляет собой *условие окончания* цикла. Оно проверяется в самом начале каждой итерации цикла. Если результатом этой инструкции является `true`, то выполнение цикла продолжается; если же результатом является `false`, цикл заканчивается. В нашем примере цикл продолжается до тех пор, пока значение переменной `index` меньше 10. На каждой итерации цикла выполняется некоторая инструкция или группа инструкций, составляющих тело цикла. В нашем случае это инструкция

```
ia[index] = index;
```

Третья управляющая инструкция цикла

```
++index
```

выполняется в конце каждой итерации, по завершении тела цикла. В нашем примере это увеличение переменной `index` на единицу. Мы могли бы записать то же действие как

```
index = index + 1
```

но C++ дает возможность использовать более короткую (и более наглядную) форму записи. Этой инструкцией завершается итерация цикла. Описанные действия повторяются до тех пор, пока условие цикла не станет ложным.

Вторая инструкция `for` в нашем примере печатает элементы массива. Она отличается от первой только тем, что в ней переменная `index` уменьшается от `9` до `0`. (Подробнее инструкция `for` рассматривается в главе 5.)

Несмотря на то, что в C++ встроена поддержка для типа данных “массив”, она весьма ограничена. Фактически мы имеем лишь возможность доступа к отдельным элементам массива. C++ не поддерживает *абстракцию массива*, не существует операций над массивами в целом, таких, например, как присвоение одного массива другому или сравнение двух массивов на равенство, и даже такой простой, на первый взгляд, операции, как получение размера массива. Мы не можем скопировать один массив в другой, используя простой оператор присваивания:

```
int array0[10]; array1[10];
...
array0 = array1; // ошибка
Вместо этого мы должны программировать такую операцию с помощью цикла:
for (int index=0; index<10; ++index)
  array0[index] = array1[index];
```

Массив “не знает” собственный размер. Поэтому мы должны сами следить за тем, чтобы случайно не обратиться к несуществующему элементу массива. Это становится особенно утомительным в таких ситуациях, как передача массива функции в качестве параметра. Можно сказать, что этот встроенный тип достался языку C++ в наследство от C и процедурно-ориентированной парадигмы программирования. В оставшейся части главы мы исследуем разные возможности “улучшить” массив.

Упражнение 2.1

Как вы думаете, почему для встроенных массивов не поддерживается операция присваивания? Какая информация нужна для того, чтобы поддержать эту операцию?

Упражнение 2.2

Какие операции должен поддерживать “полноценный” массив?

2.2. Динамическое выделение памяти и указатели

Прежде чем углубиться в объектно-ориентированную разработку, нам придется сделать небольшое отступление о работе с памятью в программе на C++. Мы не сможем написать сколько-нибудь сложную программу, не умея выделять память во время выполнения и обращаться к ней.

В C++ объекты могут быть размещены либо статически – во время компиляции, либо динамически – во время выполнения программы, путем вызова функций из стандартной библиотеки. Основная разница в использовании этих методов – в их эффективности и гибкости. Статическое размещение более эффективно, так как выделение памяти происходит *до* выполнения программы, однако оно гораздо менее гибко, потому что мы должны заранее знать тип и размер размещаемого объекта. К примеру, совсем не просто разместить содержимое некоторого текстового файла в статическом массиве строк: нам нужно заранее знать его размер. Задачи, в которых нужно хранить и обрабатывать заранее неизвестное число элементов, обычно требуют динамического выделения памяти.

До сих пор во всех наших примерах использовалось статическое выделение памяти. Скажем, определение переменной `ival`

```
int ival = 1024;
```

заставляет компилятор выделить в памяти область, достаточную для хранения переменной типа `int`, связать с этой областью имя `ival` и поместить туда значение `1024`. Все это делается на этапе компиляции, до выполнения программы.

С объектом `ival` ассоциируются две величины: собственно значение переменной, `1024` в данном случае, и адрес той области памяти, где хранится это значение. Мы можем обращаться к любой из этих двух величин. Когда мы пишем:

```
int ival2 = ival + 1;
```

то обращаемся к значению, содержащемуся в переменной ival: прибавляем к нему 1 и инициализируем переменную ival2 этим новым значением, 1025. Каким же образом обратиться к адресу, по которому размещена переменная?

C++ имеет встроенный тип "указатель", который используется для хранения адресов объектов. Чтобы объявить указатель, содержащий адрес переменной ival, мы должны написать:

```
int *pint; // указатель на объект типа int
```

Существует также специальная операция взятия адреса, обозначаемая символом &. Ее результатом является адрес объекта. Следующий оператор присваивает указателю pint

```
int *pint;
```

адрес переменной ival:

```
pint = &ival; // pint получает значение адреса ival
```

Мы можем обратиться к тому объекту, адрес которого содержит pint (ival в нашем случае), используя операцию *разыменования*, называемую также *косвенной адресацией*. Эта операция обозначается символом *. Вот как можно косвенно прибавить единицу к ival, используя ее адрес:

```
*pint = *pint + 1; // неявно увеличивает ival
```

Это выражение производит в точности те же действия, что и

```
ival = ival + 1; // явно увеличивает ival
```

В этом примере нет никакого реального смысла: использование указателя для косвенной манипуляции переменной ival менее эффективно и менее наглядно. Мы привели этот пример только для того, чтобы дать самое начальное представление об указателях. В реальности указатели используют чаще всего для манипуляций с динамически размещенными объектами.

Основные отличия между статическим и динамическим выделением памяти таковы:

- статические объекты обозначаются именованными переменными, и действия над этими объектами производятся напрямую, с использованием их имен. Динамические объекты не имеют собственных имен, и действия над ними производятся косвенно, с помощью указателей;
- выделение и освобождение памяти под статические объекты производится компилятором автоматически. Программисту не нужно самому заботиться об этом. Выделение и освобождение памяти под динамические объекты целиком и полностью возлагается на программиста. Это достаточно сложная задача, при решении которой легко наделать ошибок. Для манипуляции динамически выделяемой памятью служат операторы new и delete.

Оператор new имеет две формы. Первая форма выделяет память под единичный объект определенного типа:

```
int *pint = new int(1024);
```

Здесь оператор new выделяет память под безымянный объект типа int, инициализирует его значением 1024 и возвращает адрес созданного объекта. Этот адрес используется для инициализации указателя pint. Все действия над таким безымянным объектом производятся путем разыменовывания данного указателя, т.к. явно манипулировать динамическим объектом невозможно.

Вторая форма оператора new выделяет память под массив заданного размера, состоящий из элементов определенного типа:

```
int *pia = new int[4];
```

В этом примере память выделяется под массив из четырех элементов типа int. К сожалению, данная форма оператора new не позволяет инициализировать элементы массива.

Некоторую путаницу вносит то, что обе формы оператора new возвращают одинаковый указатель, в нашем примере это указатель на целое. И pint, и pia объявлены совершенно одинаково, однако pint указывает на единственный объект типа int, а pia – на первый элемент массива из четырех объектов типа int.

Когда динамический объект больше не нужен, мы должны явным образом освободить отведенную под него память. Это делается с помощью оператора delete, имеющего, как

```
// освобождение единичного объекта
delete pint;
// освобождение массива
```

и new, две формы – для единичного объекта и для массива:

```
delete[] pia;
```

Что случится, если мы забудем освободить выделенную память? Память будет расходоваться впустую, она окажется неиспользуемой, однако возвратить ее системе нельзя, поскольку у нас нет указателя на нее. Такое явление получило специальное название *утечка памяти*. В конце концов программа аварийно завершится из-за нехватки памяти (если, конечно, она будет работать достаточно долго). Небольшая утечка трудно поддается обнаружению, но существуют утилиты, помогающие это сделать.

Наш сжатый обзор динамического выделения памяти и использования указателей, наверное, больше породил вопросов, чем дал ответов. В разделе 8.4 затронутые проблемы будут освещены во всех подробностях. Однако мы не могли обойтись без этого отступления, так как класс Array, который мы собираемся спроектировать в последующих разделах, основан на использовании динамически выделяемой памяти.

Упражнение 2.3

```
(a) int ival = 1024;
(b) int *pi = &ival;
(c) int *pi2 = new int(1024);
```

Объясните разницу между четырьмя объектами:

```
(d) int *pi3 = new int[1024];
```

Упражнение 2.4

Что делает следующий фрагмент кода? В чем состоит логическая ошибка? (Отметим, что операция взятия индекса ([]) правильно применена к указателю pia. Объяснение этому

```
int *pi = new int(10);
int *pia = new int[10];

while ( *pi < 10 ) {
  pia[*pi] = *pi;
  *pi = *pi + 1;
}

delete pi;
```

факту можно найти в разделе 3.9.2.)

```
delete[] pia;
```

2.3. Объектный подход

В этом разделе мы спроектируем и реализуем абстракцию массива, используя механизм классов C++. Первоначальный вариант будет поддерживать только массив элементов типа int. Впоследствии при помощи шаблонов мы расширим наш массив для поддержки любых типов данных.

Первый шаг состоит в том, чтобы определить, какие операции будет поддерживать наш массив. Конечно, было бы заманчиво реализовать все мыслимые и немыслимые операции, но невозможно сделать сразу все на свете. Поэтому для начала определим то, что должен уметь наш массив:

1. обладать некоторыми знаниями о самом себе. Пусть для начала это будет знание собственного размера;
2. поддерживать операцию присваивания и операцию сравнения на равенство;
3. отвечать на некоторые вопросы, например: какова величина минимального и максимального элемента; содержит ли массив элемент с определенным значением; если да, то каков индекс первого встречающегося элемента, имеющего это значение;
4. сортировать сам себя. Пусть такая операция покажется излишней, все-таки реализуем ее в качестве дополнительного упражнения: ведь кому-то это может пригодиться.
5. Конечно, мы должны реализовать и базовые операции работы с массивом, а именно:Возможность задать размер массива при его создании. (Речь не идет о том, чтобы знать эту величину на этапе компиляции.)
6. Возможность проинициализировать массив некоторым набором значений.
7. Возможность обращаться к элементу массива по индексу. Пусть эта возможность реализуется с помощью стандартной операции взятия индекса.
8. Возможность обнаруживать обращения к несуществующим элементам массива и сигнализировать об ошибке. Не будем обращать внимание на тех потенциальных пользователей нашего класса, которые привыкли работать со встроенными массивами C и не считают данную возможность полезной – мы хотим создать такой массив, который был бы удобен в использовании даже самым неискушенным программистам на C++.

Кажется, мы перечислили достаточно потенциальных достоинств нашего будущего массива, чтобы загореться желанием немедленно приступить к его реализации. Как же это будет выглядеть на C++? В самом общем случае объявление класса выглядит

```
class classname {
public:
  // набор открытых операций
private:
  // закрытые функции, обеспечивающие реализацию
};
```

следующим образом:

`class`, `public` и `private` – это ключевые слова C++, а `classname` – имя, которое программист дал своему классу. Назовем наш проектируемый класс `IntArray`: на первом этапе этот массив будет содержать только целые числа. Когда мы научим его обращаться с данными любого типа, можно будет переименовать его в `Array`.

Определяя класс, мы создаем новый тип данных. На имя класса можно ссылаться точно так же, как на любой встроенный описатель типа. Можно создавать объекты этого нового

```
// статический объект типа IntArray
```

типа аналогично тому, как мы создаем объекты встроенных типов:

```
IntArray myArray;

// указатель на динамический объект типа IntArray
IntArray *pArray = new IntArray;
```

Определение класса состоит из двух частей: *заголовка* (имя, предваренное ключевым словом `class`) и *тела*, заключенного в фигурные скобки. Заголовок без тела может

```
// объявление класса IntArray
```

служить объявлением класса.

```
// без определения его
class IntArray;
```

Тело класса состоит из определений членов и спецификаторов доступа – ключевых слов `public`, `private` и `protected`. (Пока мы ничего не будем говорить об уровне доступа `protected`.) Членами класса могут являться функции, которые определяют набор действий, выполняемых классом, и переменные, содержащие некие внутренние данные, необходимые для реализации класса. Функции, принадлежащие классу, называют *функциями-членами* или, по-другому, *методами* класса. Вот набор методов класса `IntArray`:

```
class IntArray {
public:
  // операции сравнения: #2b
  bool operator== (const IntArray&) const;
  bool operator!= (const IntArray&) const;

  // операция присваивания: #2a
  IntArray& operator= (const IntArray&);

  int size() const; // #1
  void sort();      // #4

  int min() const;  // #3a
  int max() const;  // #3b

  // функция find возвращает индекс первого
  // найденного элемента массива
  // или -1, если элементов не найдено
  int find (int value) const; // #3c

private:
  // дальше идут закрытые члены,
  // обеспечивающие реализацию класса
  ...
}
```

Номера, указанные в комментариях при объявлениях методов, ссылаются на спецификацию класса, которую мы составили в начале данного раздела. Сейчас мы не будем объяснять смысл ключевого слова `const`, он не так уж важен для понимания того, что мы хотим продемонстрировать на данном примере. Будем считать, что это ключевое слово необходимо для правильной компиляции программы.

Именованная функция-член (например, `min()`) может быть вызвана с использованием одной из двух операций *доступа к члену класса*. Первая операция доступа, обозначаемая точкой (`.`), применяется к объектам класса, вторая – стрелка (`->`) – к указателям на объекты. Так, чтобы найти минимальный элемент в объекте, имеющем тип `IntArray`, мы

```
// инициализация переменной min_val
```

должны написать:

```
int min_val = myArray.min();
// минимальным элементом myArray
```

Чтобы найти минимальный элемент в динамически созданном объекте типа `IntArray`, мы должны написать:

```
int min_val = pArray->min();
```

(Да, мы еще ничего не сказали о том, как же проинициализировать наш объект – задать его размер и наполнить элементами. Для этого служит специальная функция-член, называемая конструктором. Мы поговорим об этом чуть ниже.)

Операции применяются к объектам класса точно так же, как и к встроенным типам данных. Пусть мы имеем два объекта типа `IntArray`:

```
IntArray myArray0, myArray1;
```

Инструкции присваивания и сравнения с этими объектами выглядят совершенно

```
// инструкция присваивания -
// вызывает функцию-член myArray0.operator=(myArray1)
myArray0 = myArray1;

// инструкция сравнения -
// вызывает функцию-член myArray0.operator==(myArray1)
if (myArray0 == myArray1)
```

обычным образом:

```
  cout << "Ура! Оператор присваивания сработал!\n";
```

Спецификаторы доступа `public` и `private` определяют уровень доступа к членам класса. К тем членам, которые перечислены после `public`, можно обращаться из любого места программы, а к тем, которые объявлены после `private`, могут обращаться только функции-члены данного класса. (Помимо функций-членов, существуют еще *функции-друзья* класса, но мы не будем говорить о них вплоть до раздела 15.2.)

В общем случае открытые члены класса составляют его открытый интерфейс, то есть набор операций, которые определяют поведение класса. Закрытые члены класса обеспечивают его *скрытую реализацию*.

Такое деление на открытый интерфейс и скрытую реализацию называют *сокрытием информации*, или *инкапсуляцией*. Это очень важная концепция программирования, мы еще поговорим о ней в следующих главах. В двух словах, эта концепция помогает решить следующие проблемы:

- если мы меняем или расширяем реализацию класса, то изменения можно выполнить так, что большинство пользовательских программ, использующих наш класс, их “не заметят”: модификации коснутся лишь скрытых членов (мы поговорим об этом в разделе 6.18);
- если в реализации класса обнаруживается ошибка, то обычно для ее исправления достаточно проверить код, составляющий именно скрытую реализацию, а не весь код программы, где данный класс используется.

Какие же внутренние данные потребуются для реализации класса `IntArray`? Необходимо где-то сохранить размер массива и сами его элементы. Мы будем хранить их в массиве встроенного типа, память для которого выделяется динамически. Так что нам потребуется

```
class IntArray {
public:
  // ...
  int size() const { return _size; }
private:
  // внутренние данные-члены
```

указатель на этот массив. Вот как будут выглядеть определения этих данных-членов:

```
  int _size;
  int *ia;
};
```

Поскольку мы поместили член _size в закрытую секцию, пользователь класса не имеет возможности обратиться к нему напрямую. Чтобы позволить внешней программе узнать размер массива, мы написали функцию-член size(), которая возвращает значение члена _size. Нам пришлось добавить символ подчеркивания к имени нашего скрытого члена _size, поскольку функция-член с именем size() уже определена. Члены класса – функции и данные – не могут иметь одинаковые имена.

Может показаться, что реализуя подобным образом доступ к скрытым данным класса, мы очень сильно проигрываем в эффективности. Сравним два выражения (предположим, что

```
IntArray array;
```

мы изменили спецификатор доступа члена _size на public):

```
int array_size = array.size();
array_size = array._size;
```

Действительно, вызов функции гораздо менее эффективен, чем прямой доступ к памяти, как во втором операторе. Так что же, принцип сокрытия информации заставляет нас жертвовать эффективностью?

На самом деле, нет. C++ имеет механизм *встроенных* (inline) функций. Текст встроенной функции подставляется компилятором в то место, где записано обращение к ней. (Это напоминает механизм макросов, реализованный во многих языках, в том числе и в C++. Однако есть определенные отличия, о которых мы сейчас говорить не будем.)

```
for (int index=0; index<array.size(); ++index)
```

Вот пример. Если у нас есть следующий фрагмент кода:

```
  // ...
```

то функция size() не будет вызываться _size раз во время исполнения. Вместо вызова компилятор подставит ее текст, и результат компиляции предыдущего кода будет в точности таким же, как если бы мы написали:

```
for (int index=0; index<array._size; ++index)
  // ...
```

Если функция определена внутри тела класса (как в нашем случае), она автоматически считается встроенной. Существует также ключевое слово inline, позволяющее объявить встроенной любую функцию3.

Мы до сих пор ничего не сказали о том, как будем инициализировать наш массив.

Одна из самых распространенных ошибок при программировании (на любом языке) состоит в том, что объект используется без предварительной инициализации. Чтобы помочь избежать этой ошибки, C++ обеспечивает механизм автоматической инициализации для определяемых пользователем классов – *конструктор* класса.

---

3 Объявление функции inline – это всего лишь подсказка компилятору. Однако компилятор не всегда может сделать функцию встроенной, существуют некоторые ограничения. Подробнее об этом сказано в разделе 7.6.

Конструктор – это специальная функция-член, которая вызывается автоматически при создании объекта типа класса. Конструктор пишется разработчиком класса, причем у одного класса может быть несколько конструкторов.

Функция-член класса, носящее то же имя, что и сам класс, считается конструктором. (Нет никаких специальных ключевых слов, позволяющих определить конструктор как-то по-другому.) Мы уже сказали, что конструкторов может быть несколько. Как же так: разные функции с одинаковыми именами?

В C++ это возможно. Разные функции могут иметь одно и то же имя, если у этих функций различны количество и/или типы параметров. Это называется *перегрузкой функции*. Обрабатывая вызов перегруженной функции, компилятор смотрит не только на ее имя, но и на список параметров. По количеству и типам передаваемых параметров компилятор может определить, какую же из одноименных функций нужно вызывать в данном случае. Рассмотрим пример. Мы можем определить следующий набор перегруженных функций `min()`. (Перегружаться могут как обычные функции, так и

```
// список перегруженных функций min()
// каждая функция отличается от других списком параметров
#include <string>

int min (const int *pia,int size);
int min (int, int);
int min (const char *str);
char min (string);
```

функции-члены.)

```
string min (string,string);
```

Поведение перегруженных функций во время выполнения ничем не отличается от поведения обычных. Компилятор определяет нужную функцию и помещает в объектный код именно ее вызов. (В главе 9 подробно обсуждается механизм перегрузки.)

Итак, вернемся к нашему классу `IntArray`. Давайте определим для него три

```
class IntArray {
public:
  explicit IntArray (int sz = DefaultArraySize);
  IntArray (int *array, int array_size);
  IntArray (const IntArray &rhs);
  // ...
private:
  static const int DefaultArraySize = 12;
```

конструктора:

```
}
```

Первый из перечисленных конструкторов

```
IntArray (int sz = DefaultArraySize);
```

называется *конструктором по умолчанию*, потому что он может быть вызван без параметров. (Пока не будем объяснять ключевое слово `explicit`.) Если при создании объекта ему задается параметр типа `int`, например

```
IntArray array1(1024);
```

то значение 1024 будет передано в конструктор. Если же размер не задан, допустим:

```
IntArray array2;
```

то в качестве значения отсутствующего параметра конструктор принимает величину DefaultArraySize. (Не будем пока обсуждать использование ключевого слова static в определении члена DefaultArraySize: об этом говорится в разделе 13.5. Скажем лишь, что такой член данных существует в единственном экземпляре и принадлежит одновременно *всем* объектам данного класса.)

```
IntArray::IntArray (int sz)
{
  // инициализация членов данных
  _size = sz;
  ia = new int[_size];

  // инициализация элементов массива
  for (int ix=0; ix<_size; ++ix)
    ia[ix] = 0;
```

Вот как может выглядеть определение нашего конструктора по умолчанию:

```
}
```

Это определение содержит несколько упрощенный вариант реализации. Мы не позаботились о том, чтобы попытаться избежать возможных ошибок во время выполнения. Какие ошибки возможны? Во-первых, оператор new может потерпеть неудачу при выделении нужной памяти: в реальной жизни память не бесконечна. (В разделе 2.6 мы увидим, как обрабатываются подобные ситуации.) А во-вторых, параметр sz из-за небрежности программиста может иметь некорректное значение, например нуль или отрицательное.

Что необычного мы видим в таком определении конструктора? Сразу бросается в глаза первая строчка, в которой использована *операция разрешения области видимости* (::):

```
IntArray::IntArray(int sz);
```

Дело в том, что мы определяем нашу функцию-член (в данном случае конструктор) вне тела класса. Для того чтобы показать, что эта функция на самом деле является членом класса IntArray, мы должны явно предварить имя функции именем класса и двойным двоеточием. (Подробно области видимости разбираются в главе 8; области видимости применительно к классам рассматриваются в разделе 13.9.)

Второй конструктор класса IntArray инициализирует объект IntArray значениями элементов массива встроенного типа. Он требует двух параметров: массива встроенного типа со значениями для инициализации и размера этого массива. Вот как может

```
int ia[10] = {0,1,2,3,4,5,6,7,8,9};
```

выглядеть создание объекта IntArray с использованием данного конструктора:

```
IntArray iA3(ia,10);
```

Реализация второго конструктора очень мало отличается от реализации конструктора по

```
IntArray::IntArray (int *array, int sz)
{
  // инициализация членов данных
  _size = sz;
  ia = new int[_size];

  // инициализация элементов массива
  for (int ix=0; ix<_size; ++ix)
    ia[ix] = array[ix];
}
```

умолчанию. (Как и в первом случае, мы пока опустили обработку ошибочных ситуаций.)

Третий конструктор называется *копирующим конструктором*. Он инициализирует один объект типа `IntArray` значением другого объекта `IntArray`. Такой конструктор вызывается автоматически при выполнении следующих инструкций:

```
IntArray array;

// следующие два объявления совершенно эквивалентны:
IntArray ia1 = array;
IntArray ia2 (array);
```

Вот как выглядит реализация копирующего конструктора для `IntArray`, опять-таки без

```
IntArray::IntArray (const IntArray &rhs )
```

обработки ошибок:

```
{
  // инициализация членов данных
  _size = rhs._size;
  ia = new int[_size];

  // инициализация элементов массива
  for (int ix=0; ix<_size; ++ix)
    ia[ix] = rhs.ia[ix];
}
```

В этом примере мы видим еще один составной тип данных – *ссылку* на объект, которая обозначается символом &. Ссылку можно рассматривать как разновидность указателя: она также позволяет косвенно обращаться к объекту. Однако синтаксис их использования различается: для доступа к члену объекта, на который у нас есть ссылка, следует использовать точку, а не стрелку; следовательно, мы пишем `rhs._size`, а не `rhs->_size`. (Ссылки рассматриваются в разделе 3.6.)

Заметим, что реализация всех трех конструкторов очень похожа. Если один и тот же код повторяется в разных местах, желательно вынести его в отдельную функцию. Это облегчает и дальнейшую модификацию кода, и чтение программы. Вот как можно модернизировать наши конструкторы, если выделить повторяющийся код в отдельную функцию `init()`:

```
class IntArray {
public:
  explicit IntArray (int sz = DefaultArraySize);
  IntArray (int *array, int array_size);
  IntArray (const IntArray &rhs);
  // ...
private:
  void init (int sz,int *array);
  // ...
};

// функция, используемая всеми конструкторами
void IntArray::init (int sz,int *array)
{
  _size = sz;
  ia = new int[_size];

  for (int ix=0; ix<_size; ++ix)
    if ( !array )
      ia[ix] = 0;
    else
      ix[ix] = array[ix];
}

// модифицированные конструкторы
IntArray::IntArray (int sz) { init(sz,0); }
IntArray::IntArray (int *array, int array_size)
     { init (array_size,array); }
IntArray::IntArray (const IntArray &rhs)
     { init (rhs._size,rhs.ia); }
```

Имеется еще одна специальная функция-член – *деструктор*, который автоматически вызывается в тот момент, когда объект прекращает существование. Имя деструктора совпадает с именем класса, только в начале идет символ тильды (~). Основное назначение данной функции – освободить ресурсы, отведенные объекту во время его создания и использования. Применение деструкторов помогает бороться с трудно обнаруживаемыми ошибками, ведущими к утечке памяти и других ресурсов. В случае класса `IntArray` эта функция-член должна освободить память, выделенную в момент создания объекта. (Подробно конструкторы и деструкторы описаны в главе 14.) Вот как выглядит

```
class IntArray {
```

деструктор для `IntArray`:

```
public:
  // конструкторы
  explicit IntArray (int sz = DefaultArraySize);
  IntArray (int *array, int array_size);
  IntArray (const IntArray &rhs);

  // деструктор
  ~IntArray() { delete[] ia; }
  // ...
private:
  // ...
};
```

Теперь нам нужно определить операции доступа к элементам массива `IntArray`. Мы хотим, чтобы обращение к элементам `IntArray` выглядело точно так же, как к элементам

```
IntArray array;
```

массива встроенного типа, с использованием оператора взятия индекса:

```
int last_pos = array.size()-1;

int temp = array[0];
array[0] = array[last_pos];
array[last_pos] = temp;
```

Для реализации доступа мы используем возможность *перегрузки операций*. Вот как

```
#include <cassert>

int& IntArray::operator[] (int index)
{
  assert (index >= 0 && index < _size);
  return ia[index];
```

выглядит функция, реализующая операцию взятия индекса:

```
}
```

Обычно для проектируемого класса перегружают операции присваивания, операцию сравнения на равенство, возможно, операции сравнения по величине и операции ввода/вывода. Как и перегруженных функций, перегруженных операторов, отличающихся типами операндов, может быть несколько. К примеру, можно создать несколько операций присваивания объекту значения другого объекта того же самого или иного типа. Конечно, эти объекты должны быть более или менее “похожи”. (Подробно о перегрузке операций мы расскажем в главе 15, а в разделе 3.15 приведем еще несколько примеров.)

Определения класса, различных относящихся к нему констант и, быть может, каких-то еще переменных и макросов по принятым соглашениям помещаются в заголовочный файл, имя которого совпадает с именем класса. Для класса `IntArray` мы должны создать заголовочный файл `IntArray.h`. Любая программа, в которой будет использоваться класс `IntArray`, должна включать этот заголовочный файл директивой препроцессора `#include`.

По тому же самому соглашению функции-члены класса, определенные вне его описания, помещаются в файл с именем класса и расширением, обозначающим исходный текст C++ программы. Мы будем использовать расширение `.C` (напомним, что в разных системах вы можете встретиться с разными расширениями исходных текстов C++ программ) и назовем наш файл `IntArray.C`.

Упражнение 2.5

Ключевой особенностью класса C++ является разделение интерфейса и реализации. Интерфейс представляет собой набор операций (функций), выполняемых объектом; он определяет имя функции, возвращаемое значение и список параметров. Обычно пользователь не должен знать об объекте ничего, кроме его интерфейса. Реализация скрывает алгоритмы и данные, нужные объекту, и может меняться при развитии объекта,

никак не затрагивая интерфейс. Попробуйте определить интерфейсы для одного из следующих классов (выберите любой):

(a) матрица

(b) булевское значение

(c) паспортные данные человека

(d) дата

(e) указатель

(f) точка

Упражнение 2.6

Попробуйте определить набор конструкторов, необходимых для класса, выбранного вами в предыдущем упражнении. Нужен ли деструктор для вашего класса? Помните, что на самом деле конструктор не создает объект: память под объект отводится до начала работы данной функции, и конструктор только производит определенные действия по инициализации объекта. Аналогично деструктор уничтожает не сам объект, а только те дополнительные ресурсы, которые могли быть выделены в результате работы конструктора или других функций-членов класса.

Упражнение 2.7

В предыдущих упражнениях вы практически полностью определили интерфейс выбранного вами класса. Попробуйте теперь написать программу, использующую ваш класс. Удобно ли пользоваться вашим интерфейсом? Не хочется ли Вам пересмотреть спецификацию? Сможете ли вы сделать это и одновременно сохранить совместимость со старой версией?

2.4. Объектно-ориентированный подход

Вспомним спецификацию нашего массива в предыдущем разделе. Мы говорили о том, что некоторым пользователям может понадобиться упорядоченный массив, в то время как большинство, скорее всего, удовлетворится и неупорядоченным. Если представить себе, что наш массив `IntArray` упорядочен, то реализация таких функций, как `min()`, `max()`, `find()`, должна отличаться от их реализации для массива неупорядоченного большей эффективностью. Вместе с тем, для поддержания массива в упорядоченном состоянии все прочие функции должны быть сильно усложнены.

Мы выбрали наиболее общий случай – неупорядоченный массив. Но как же быть с теми немногочисленными пользователями, которым обязательно нужна функциональность массива упорядоченного? Мы должны специально для них создать другой вариант массива?

А вот и еще одна категория недовольных пользователей: их не удовлетворяют накладные расходы на проверку правильности индекса. Мы исходили из того, что корректность работы нашего класса превыше всего, и старались обезопасить себя от ошибочных ситуаций. Но возьмем, к примеру, разработчиков систем виртуальной реальности. Трехмерные изображения должны строиться с максимально возможной скоростью, быть может, за счет точности.

Да, мы можем удовлетворить и тех и других, создав для каждой группы пользователей свой, немного модернизированный, вариант `IntArray`. Более того, его даже не слишком трудно сделать, поскольку мы старались создать хорошую реализацию и необходимые

изменения затронут совсем небольшие участки кода. Итак, копируем исходный текст,

```
// неупорядоченный массив без проверки границ индекса
class IntArray { ... };

// неупорядоченный массив с проверкой границ индекса
class IntArrayRC { ... };

// упорядоченный массив без проверки границ индекса
```

вносим необходимые изменения в нужные места и получаем три класса:

```
class IntSortedArray { ... };
```

Подобное решение имеет следующие недостатки:

- нам необходимо сопровождать три копии кода, различающиеся весьма незначительно. Хорошо бы выделить общие участки кода. Кроме упрощения сопровождения, это позволит использовать их впоследствии, если мы захотим создать еще один вариант массива, например упорядоченный с проверкой границ индекса;
- если понадобится какая-то общая функция для обработки всех наших массивов, то нам придется написать три копии, поскольку типы ее параметров будут

  ```
  void process_array (IntArray&);
  void process_array (IntArrayRC&);
  ```

  различаться:

  ```
  void process_array (IntSortedArray&);
  ```

  хотя реализация этих функций может быть совершенно идентичной. Было бы лучше написать единственную функцию, которая могла бы работать не только со всеми нашими массивами, но и с теми их вариациями, какие мы, возможно, реализуем впоследствии.

Парадигма объектно-ориентированного программирования позволяет осуществить все эти пожелания. Механизм *наследования* обеспечивает пожелания из первого пункта. Если один класс является потомком другого (например, `IntArrayRC` потомок класса `IntArray`), то наследник имеет возможность пользоваться всеми данными и функциями-членами, определенными в классе-предке. То есть класс `IntArrayRC` может просто использовать всю основную функциональность, предоставляемую классом `IntArray`, и добавить только то, что нужно ему для обеспечения проверки границ индекса.

В C++ класс, свойства которого наследуются, называют также *базовым классом*, а класс-наследник – *производным классом*, или *подклассом* базового. Класс и подкласс имеют общий интерфейс, предоставляемый базовым классом (т.к. подкласс имеет все функции-члены базового класса). Значит, программу, использующую только функции из этого общего интерфейса, не должен интересовать фактический тип объекта, с которым она работает, – базового ли типа этот объект или производного. В этом смысле общий интерфейс скрывает специфичные для подкласса детали. Отношения между классами и подклассами называются *иерархией наследования классов*. Вот как может выглядеть реализация функции `swap()`, которая меняет местами два указанных элемента массива. Первым параметром функции является ссылка на базовый класс `IntArray`:

```
#include <IntArray.h>

void swap (IntArray &ia, int i, int j)
{
  int temp ia[i];
  ia[i] = ia[j];
  ia[j] = temp;
}

// ниже идут обращения к функции swap:
IntArray ia;
IntArrayRC iarc;
IntSortedArray ias;
// правильно - ia имеет тип IntArray
swap (ia,0,10);

// правильно - iarc является подклассом IntArray
swap (iarc,0,10);

// правильно - ias является подклассом IntArray
swap (ias,0,10);
```

```
// ошибка - string не является подклассом IntArray
string str("Это не IntArray!");
swap (str,0,10);
```

Каждый из трех классов реализует операцию взятия индекса по-своему. Поэтому важно, чтобы внутри функции `swap()` вызывалась нужная операция взятия индекса. Так, если `swap()` вызвана для `IntArrayRC`:

```
swap (iarc,0,10);
```

то должна вызываться функция взятия индекса для объекта класса `IntArrayRC`, а для

```
swap (ias,0,10);
```

функция взятия индекса `IntSortedArray`. Именно это и обеспечивает механизм *виртуальных* функций C++.

Давайте попробуем сделать наш класс `IntArray` базовым для иерархии подклассов. Что нужно изменить в его описании? Синтаксически – совсем немного. Возможно, придется открыть для производных классов доступ к скрытым членам класса. Кроме того, те функции, которые мы собираемся сделать виртуальными, необходимо явно пометить специальным ключевым словом `virtual`. Основная же трудность состоит в таком изменении реализации базового класса, которая позволит ей лучше отвечать своей новой цели – служить базой для целого семейства подклассов.

При простом объектном подходе можно выделить двух разработчиков конечной программы – разработчик класса и пользователь класса (тот, кто использует данный класс в конечной программе), причем последний обращается только к открытому интерфейсу. Для такого случая достаточно двух уровней доступа к членам класса – *открытого* (`public`) и *закрытого* (`private`).

Если используется наследование, то к этим двум группам разработчиков добавляется третья, промежуточная. Производный класс может проектировать совсем не тот человек, который проектировал базовый, и для того чтобы реализовать класс-наследник, совсем не

обязательно иметь доступ к реализации базового. И хотя такой доступ может потребоваться при проектировании подкласса, от конечного пользователя обоих классов эта часть по-прежнему должна быть закрыта. К двум уровням доступа добавляется третий, в некотором смысле промежуточный, – *защищенный* (protected). Члены класса, объявленные как защищенные, могут использоваться классами-потомками, но никем больше. (Закрытые члены класса недоступны даже для его потомков.)

```
class IntArray {
public:
  // конструкторы
  explicit IntArray (int sz = DefaultArraySize);
  IntArray (int *array, int array_size);
  IntArray (const IntArray &rhs);

  // виртуальный деструктор
  virtual ~IntArray() { delete[] ia; }

  // операции сравнения:
  bool operator== (const IntArray&) const;
  bool operator!= (const IntArray&) const;

  // операция присваивания:
  IntArray& operator= (const IntArray&);
  int size() const { return _size; };

  // мы убрали проверку индекса...
```

Вот как выглядит модифицированное описание класса IntArray:

```
  virtual int& operator[](int index)
       { return ia[index]; }
  virtual void sort();

  virtual int min() const;
  virtual int max() const;
  virtual int find (int value) const;

protected:
  static const int DefaultArraySize = 12;
  void init (int sz; int *array);

  int _size;
  int *ia;
}
```

Открытые функции-члены по-прежнему определяют интерфейс класса, как и в реализации из предыдущего раздела. Но теперь это интерфейс не только базового, но и всех производных от него подклассов.

Нужно решить, какие из членов, ранее объявленных как закрытые, сделать защищенными. Для нашего класса IntArray сделаем защищенными все оставшиеся члены.

Теперь нам необходимо определить, реализация каких функций-членов базового класса может меняться в подклассах. Такие функции мы объявим виртуальными. Как уже отмечалось выше, реализация операции взятия индекса будет отличаться по крайней мере для подкласса IntArrayRC. Реализация операторов сравнения и функции size() одинакова для всех подклассов, следовательно, они не будут виртуальными.

При вызове невиртуальной функции компилятор определяет все необходимое еще на этапе компиляции. Если же он встречает вызов виртуальной функции, то не пытается сделать этого. Выбор нужной из набора виртуальных функций (*разрешение вызова*) происходит во время выполнения программы и основывается на типе объекта, из

```
void init (IntArray &ia)
```

которого она вызвана. Рассмотрим пример:

```
{
  for (int ix=0; ix<ia.size(); ++ix)
    ia[ix] = ix;
}
```

Формальный параметр функции `ia` может быть ссылкой на `IntArray`, `IntArrayRC` или на `IntSortedArray`. Функция-член `size()` не является виртуальной и разрешается на этапе компиляции. А вот виртуальный оператор взятия индекса не может быть разрешен на данном этапе, поскольку реальный тип объекта, на который ссылается `ia`, в этот момент неизвестен.

(В главе 17 мы будем говорить о виртуальных функциях более подробно. Там мы рассмотрим также и накладные расходы, которые влечет за собой их использование.)

```
#ifndef IntArrayRC_H
#define IntArrayRC_H

#include "IntArray.h"

class IntArrayRC : public IntArray {
public:
    IntArrayRC( int sz = DefaultArraySize );
    IntArrayRC( const int *array, int array_size );
    IntArrayRC( const IntArrayRC &rhs );

    virtual int& operator[]( int ) const;

private:
    void check_range( int ix );
};
```

Вот как выглядит определение производного класса `IntArrayRC`:

```
#endif
```

Этот текст мы поместим в заголовочный файл `IntArrayRC.h`. Обратите внимание на то, что в наш файл включен заголовочный файл `IntArray.h`.

В классе `IntArrayRC` мы должны реализовать только те особенности, которые отличают его от `IntArray`: класс `IntArrayRC` должен иметь свою собственную реализацию операции взятия индекса; функцию для проверки индекса и собственный набор конструкторов.

Все данные и функции-члены класса `IntArray` можно использовать в классе `IntArrayRC` так, как будто это его собственные члены. В этом и заключается смысл наследования. Синтаксически наследование выражается строкой

```
class IntArrayRC : public IntArray
```

Эта строка показывает, что класс IntArrayRC произведен от класса IntArray, другими словами, наследует ему. Ключевое слово public в данном контексте говорит о том, что производный класс сохраняет открытый интерфейс базового класса, то есть что все открытые функции базового класса остаются открытыми и в производном. Объект типа IntArrayRC может использоваться вместо объекта типа IntArray, как, например, в приведенном выше примере с функцией swap(). Таким образом, подкласс IntArrayRC – это расширенная версия класса IntArray.

```
IntArrayRC::operator[]( int index )
{
    check_range( index );
    return _ia[ index ];
```

Вот как выглядит реализация операции взятия индекса:

```
}
```

```
#include <cassert>

inline void IntArrayRC::check_range(int index)
{
    assert (index>=0 && index < _size);
```

А вот реализация встроенной функции check_range():

```
}
```

(Мы говорили о макросе assert() в разделе 1.3.)

Почему проверка индекса вынесена в отдельную функцию, а не выполняется прямо в теле оператора взятия индекса? Потому что, если мы когда-нибудь потом захотим изменить что-то в реализации проверки, например написать свою обработку ошибок, а не использовать assert(), это будет сделать проще.

В каком порядке активизируются конструкторы при создании производного класса? Первым вызывается конструктор базового класса, инициализирующий те члены, которые входят в базовый класс. Затем начинает работать конструктор производного класса, где мы должны проинициализировать только те члены, которые являются специфичными для подкласса, то есть отсутствуют в базовом классе.

Однако заметим, что в нашем производном классе IntArrayRC нет новых членов, представляющих данные. Значит ли это, что нам не нужно реализовывать конструкторы для него? Ведь вся работа по инициализации членов данных уже проделана конструкторами базового класса.

На самом деле конструкторы, как и деструкторы или операторы присваивания, не наследуются – это правило языка C++. Кроме того, конструктор производного класса обеспечивает механизм передачи параметров конструктору базового класса. Рассмотрим

```
int ia[] = {0,1,1,2,3,5,8,13};
```

пример. Пусть мы хотим создать объект класса IntArrayRC следующим образом:

```
IntArrayRC iarc(ia,8);
```

Нам нужно передать параметры `ia` и `8` конструктору базового класса `IntArray`. Для этого служит специальная синтаксическая конструкция. Вот как выглядят реализации

```
inline IntArrayRC::IntArrayRC( int sz )
    : IntArray( sz ) {}

inline IntArrayRC::IntArrayRC( const int *iar, int sz )
```

двух конструкторов `IntArrayRC`:

```
    : IntArray( iar, sz ) {}
```

(Мы будем подробно говорить о конструкторах в главах 14 и 17. Там же мы покажем, почему не нужно реализовывать конструктор копирования для `IntArrayRC`.)

Часть определения, следующая за двоеточием, называется *списком инициализации членов*. Именно здесь, указав конструктор базового класса, мы можем передать ему параметры. Тела обоих конструкторов пусты, поскольку их работа состоит исключительно в передаче параметров конструктору базового класса. Нам не нужно реализовывать деструктор для `IntArrayRC`, так как ему просто нечего делать. Точно так же, как при создании объекта производного типа вызывается сначала конструктор базового типа, а затем производного, при уничтожении автоматически вызываются деструкторы – естественно, в обратном порядке: сначала деструктор производного, затем базового. Таким образом, деструктор базового класса будет вызван для объекта типа `IntArrayRC`, хотя тот и не имеет собственной аналогичной функции.

Мы поместим все встроенные функции класса `IntArrayRC` в тот же заголовочный файл `IntArrayRC.h`. Поскольку у нас нет невстроенных функций, то создавать файл `IntArrayRC.C` не нужно.

Вот пример простой программы, использующей классы `IntArray` и `IntArrayRC`:

```
#include <iostream>
#include "IntArray.h"
#include "IntArrayRC.h"

void swap( IntArray &ia, int ix, int jx )
{
  int tmp  = ia[ ix ];
  ia[ ix ] = ia[ jx ];
  ia[ jx ] = tmp;
}

int main()
{
  int array[ 4 ] = { 0, 1, 2, 3 };
  IntArray ia1( array, 4 );
  IntArrayRC ia2( array, 4 );

  // ошибка: должно быть size-1
  // не может быть выявлена объектом IntArray
  cout << "swap() with IntArray ia1" << endl;
  swap( ia1, 1, ia1.size() );

  // правильно: объект IntArrayRC "поймает" ошибку
  cout << "swap() with IntArrayRC ia2" << endl;
  swap( ia2, 1, ia2.size() );
  return 0;
}
```

При выполнении программа выдаст следующий результат:

```
swap() with IntArray ia1
swap() with IntArrayRC ia2
Assertion failed: ix >= 0 && ix < _size,
  file IntArrayRC.h, line 19
```

Упражнение 2.8

Отношение наследования между типом и подтипом служит примером отношения *является*. Так, массив `IntArrayRC` является подвидом массива `IntArray`, книга является подвидом выдаваемых библиотекой предметов, аудиокнига является подвидом книги и

(a) функция-член является подвидом функции

(b) функция-член является подвидом класса

(c) конструктор является подвидом функции-члена

(d) самолет является подвидом транспортного средства

(e) машина является подвидом грузовика

(f) круг является подвидом геометрической фигуры

(g) квадрат является подвидом треугольника

(h) автомобиль является подвидом самолета

т.д. Какие из следующих утверждений верны?

(i) читатель является подвидом библиотеки

Упражнение 2.9

Определите, какие из следующих функций могут различаться в реализации для

(a) `rotate();`

(b) `print();`

(c) `size();`

(d) `DateBorrowed(); // дата выдачи книги`

(e) `rewind();`

(f) `borrower(); // читатель`

(g) `is_late(); // книга просрочена`

производных классов и, таким образом, выступают кандидатами в виртуальные функции:

(h) `is_on_loan(); // книга выдана`

Упражнение 2.10

Ходят споры о том, не нарушает ли принципа инкапсуляции введение защищенного уровня доступа. Есть мнение, что для соблюдения этого принципа следует отказаться от использования такого уровня и работать только с закрытыми членами. Противоположная точка зрения гласит, что без защищенных членов производные классы невозможно реализовывать достаточно эффективно и в конце концов пришлось бы везде задействовать открытый уровень доступа. А каково ваше мнение по этому поводу?

Упражнение 2.11

Еще одним спорным аспектом является необходимость явно указывать виртуальность функций в базовом классе. Есть мнение, что все функции должны быть виртуальными по умолчанию, тогда ошибка в разработке базового класса не повлечет таких серьезных последствий в разработке производного, когда из-за невозможности изменить реализацию функции, ошибочно не определенной в базовом классе как виртуальная, приходится сильно усложнять реализацию. С другой стороны, виртуальные функции невозможно объявить как встроенные, и использование только таких функций сильно снизит эффективность. Каково ваше мнение?

Упражнение 2.12

Каждая из приведенных ниже абстракций определяет целое семейство подвидов, как, например, абстракция “транспортное средство” может определять “самолет”, “автомобиль”, “велосипед”. Выберите одно из семейств и составьте для него иерархию подвидов. Приведите пример открытого интерфейса для этой иерархии, включая конструкторы. Определите виртуальные функции. Напишите псевдокод маленькой

(a) Точка

(b) Служащий

(c) Фигура

(d) Телефонный_номер

(e) Счет_в_банке

программы, использующей данный интерфейс.

(f) Курс_продажи

2.5. Использование шаблонов

Наш класс `IntArray` служит хорошей альтернативой встроенному массиву целых чисел. Но в жизни могут потребоваться массивы для самых разных типов данных. Можно предположить, что единственным отличием массива элементов типа `double` от нашего является тип данных в объявлениях, весь остальной код совпадает буквально.

Для решения данной проблемы в C++ введен механизм *шаблонов*. В объявлениях классов и функций допускается использование *параметризованных* типов. Типы-параметры заменяются в процессе компиляции настоящими типами, встроенными или определенными пользователем. Мы можем создать шаблон класса `Array`, заменив в классе `IntArray` тип элементов `int` на обобщенный тип-параметр. Позже мы *конкретизируем* типы-параметры, подставляя вместо них реальные типы `int`, `double` и `string`. В результате появится способ использовать эти конкретизации так, как будто мы на самом деле определили три разных класса для этих трех типов данных.

```
template <class elemType>
class Array {
public:
    explicit Array( int sz = DefaultArraySize );
    Array( const elemType *ar, int sz );
    Array( const Array &iA );

    virtual ~Array() { delete[] _ia; }

    Array& operator=( const Array & );
    int size() const { return _size; }

    virtual elemType& operator[]( int ix )
        { return _ia[ix]; }

    virtual void sort( int,int );
    virtual int find( const elemType& );
    virtual elemType min();
    virtual elemType max();
protected:
    void init( const elemType*, int );
    void swap( int, int );
    static const int DefaultArraySize = 12;
    int       _size;
    elemType *_ia;
```

Вот как может выглядеть шаблон класса `Array`:

```
};
```

Ключевое слово `template` говорит о том, что задается шаблон, параметры которого заключаются в угловые скобки (<>). В нашем случае имеется лишь один параметр `elemType`; ключевое слово `class` перед его именем сообщает, что этот параметр представляет собой тип.

При конкретизации класса-шаблона `Array` параметр `elemType` заменяется на реальный тип при каждом использовании, как показано в примере:

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

int main()
{
  const int array_size = 4;

  // elemType заменяется на int
  Array<int> ia(array_size);

  // elemType заменяется на double
  Array<double> da(array_size);

  // elemType заменяется на char
  Array<char> ca(array_size);

  int ix;

  for ( ix = 0; ix < array_size; ++ix ) {
          ia[ix] = ix;
          da[ix] = ix * 1.75;
          ca[ix] = ix + 'a';
  }

  for ( ix = 0; ix < array_size; ++ix )
          cout << "[ " << ix << " ]  ia: "  << ia[ix]
              << "\tca: " << ca[ix]
              << "\tda: " << da[ix] << endl;

  return 0;
}
```

```
Array<int> ia(array_size);
```

Здесь определены три экземпляра класса `Array`:

```
Array<double> da(array_size);
Array<char> ca(array_size);
```

Что делает компилятор, встретив такое объявление? Подставляет текст шаблона `Array`, заменяя параметр `elemType` на тот тип, который указан в каждом конкретном случае. Следовательно, объявления членов приобретают в первом случае такой вид:

```
// Array<int> ia(array_size);
int _size;
int *_ia;
```

Заметим, что это в точности соответствует определению массива `IntArray`.

```
// Array<double> da(array_size);
```

Для оставшихся двух случаев мы получим следующий код:

```
int     _size;
double *_ia;
```

```
// Array<char> ca(array_size);
int   _size;
char *_ia;
```

Что происходит с функциями-членами? В них тоже тип-параметр `elemType` заменяется на реальный тип, однако компилятор не конкретизирует те функции, которые не вызываются в каком-либо месте программы. (Подробнее об этом в разделе 16.8.)

При выполнении программа этого примера выдаст следующий результат:

```
[ 0 ]  ia: 0    ca: a   da: 0
[ 1 ]  ia: 1    ca: b   da: 1.75
[ 2 ]  ia: 2    ca: c   da: 3.5
[ 3 ]  ia: 3    ca: d   da: 5.25
```

Механизм шаблонов можно использовать и в наследуемых классах. Вот как выглядит

```
#include <cassert>
```

определение шаблона класса `ArrayRC`:

```
#include "Array.h"

template <class elemType>
class ArrayRC : public Array<elemType> {
public:
    ArrayRC( int sz = DefaultArraySize )
            : Array<elemType>( sz ) {}
    ArrayRC( const ArrayRC& r )
            : Array<elemType>( r ) {}
    ArrayRC( const elemType *ar, int sz )
            : Array<elemType>( ar, sz ) {}

    elemType& ArrayRC<elemType>::operator[]( int ix )
    {
        assert( ix >= 0 && ix < Array<elemType>::_size );
        return _ia[ ix ];
    }
private:
// ...
};
```

Подстановка реальных параметров вместо типа-параметра `elemType` происходит как в базовом, так и в производном классах. Определение

```
ArrayRC<int> ia_rc(10);
```

ведет себя точно так же, как определение `IntArrayRC` из предыдущего раздела. Изменим

```
// функцию swap() тоже следует сделать шаблоном
```

пример использования из предыдущего раздела. Прежде всего, чтобы оператор

```
swap( ia1, 1, ia1.size() );
```

был допустимым, нам потребуется представить функцию `swap()` в виде шаблона.

```
#include "Array.h"
```

```
template <class elemType>
inline void
swap( Array<elemType> &array, int i, int j )
{
  elemType tmp = array[ i ];
  array[ i ] = array[ j ];
  array[ j ] = tmp;
}
```

При каждом вызове `swap()` генерируется подходящая конкретизация, которая зависит от

```
#include <iostream>
```

типа массива. Вот как выглядит программа, использующая шаблоны `Array` и `ArrayRC`:

```
#include "Array.h"
#include "ArrayRC.h"

template <class elemType>
inline void
swap( Array<elemType> &array, int i, int j )
{
  elemType tmp = array[ i ];
  array[ i ] = array[ j ];
  array[ j ] = tmp;
}

int main()
{
  Array<int>   ia1;
  ArrayRC<int> ia2;

  cout << "swap() with Array<int> ia1" << endl;
  int size = ia1.size();
  swap( ia1, 1, size );

  cout << "swap() with ArrayRC<int> ia2" << endl;
  size = ia2.size();
  swap( ia2, 1, size );

  return 0;
}
```

Упражнение 2.13

```
template<class elemType> class Array;
```

Пусть мы имеем следующие объявления типов:

```
enum Status { ... };
typedef string *Pstring;
```

Есть ли ошибки в приведенных ниже описаниях объектов?

(a) `Array< int*& > pri(1024);`

(b) `Array< Array<int> > aai(1024);`

(c) `Array< complex< double > > acd(1024);`

(d) `Array< Status > as(1024);`

(e) `Array< Pstring > aps(1024);`

Упражнение 2.14

```
class example1 {
public:
  example1 (double min, double max);
  example1 (const double *array, int size);

  double& operator[] (int index);
  bool operator== (const example1&) const;

  bool insert (const double*, int);
  bool insert (double);

  double min (double) const { return _min; };
  double max (double) const { return _max; };

  void min (double);
  void max (double);

  int count (double value) const;

private:
  int size;
  double *parray;
  double _min;
  double _max;
```

Перепишите следующее определение, сделав из него шаблон класса:

```
}
```

Упражнение 2.15

```
template <class elemType> class Example2 {
```

Имеется следующий шаблон класса:

```
public:
  explicit Example2 (elemType val=0) : _val(val) {};

  bool min(elemType value) { return _val < value; }
  void value(elemType new_val) { _val = new_val; }
  void print (ostream &os) { os << _val; }

private:
  elemType _val;
}

template <class elemType>
```

```
ostream& operator<<(ostream &os,const Example2<elemType> &ex)
  { ex.print(os); return os; }
```

(a) `Example2<Array<int>*> ex1;`

(b) `ex1.min (&ex1);`

(c) `Example2<int> sa(1024),sb;`

(d) `sa = sb;`

(e) `Example2<string> exs("Walden");`

Какие действия вызывают следующие инструкции?

(f) `cout << "exs: " << exs << endl;`

Упражнение 2.16

Пример из предыдущего упражнения накладывает определенные ограничения на типы данных, которые могут быть подставлены вместо `elemType`. Так, параметр конструктора имеет по умолчанию значение `0`:

```
explicit Example2 (elemType val=0) : _val(val) {};
```

Однако не все типы могут быть инициализированы нулем (например, тип `string`), поэтому определение объекта

```
Example2<string> exs("Walden");
```

является правильным, а

```
Example2<string> exs2;
```

приведет к синтаксической ошибке[4]. Также ошибочным будет вызов функции `min()`, если для данного типа не определена операция *меньше*. C++ не позволяет задать ограничения для типов, подставляемых в шаблоны. Как вы думаете, было бы полезным иметь такую возможность? Если да, попробуйте придумать синтаксис задания ограничений и перепишите в нем определение класса `Example2`. Если нет, поясните почему.

Упражнение 2.17

Как было показано в предыдущем упражнении, попытка использовать шаблон `Example2` с типом, для которого не определена операция *меньше*, приведет к синтаксической ошибке. Однако ошибка проявится только тогда, когда в тексте компилируемой программы действительно встретится вызов функции `min()`, в противном случае компиляция пройдет успешно. Как вы считаете, оправдано ли такое поведение? Не лучше ли предупредить об ошибке сразу, при обработке описания шаблона? Поясните свое мнение.

---

4 Вот как выглядит общее решение этой проблемы:

```
Example2( elemType nval = elemType() ) " _val( nval ) {}
```

2.6. Использование исключений

Исключениями называют аномальные ситуации, возникающие во время исполнения программы: невозможность открыть нужный файл или получить необходимое количество памяти, использование выходящего за границы индекса для какого-либо массива. Обработка такого рода исключений, как правило, плохо интегрируется в основной алгоритм программы, и программисты вынуждены изобретать разные способы корректной обработки исключения, стараясь в то же время не слишком усложнить программу добавлением всевозможных проверок и дополнительных ветвей алгоритма.

C++ предоставляет стандартный способ реакции на исключения. Благодаря вынесению в отдельную часть программы кода, ответственного за проверку и обработку ошибок, значительно облегчается восприятие текста программы и сокращается ее размер. Единый синтаксис и стиль обработки исключений можно, тем не менее, приспособить к самым разнообразным нуждам и запросам.

Механизм исключений делится на две основные части:

точка программы, в которой произошло исключение. Определение того факта, что при выполнении возникла какая-либо ошибка, влечет за собой *возбуждение* исключения. Для этого в C++ предусмотрен специальный оператор `throw`. Возбуждение исключения в

```
if ( !infile ) {
  string errMsg("Невозможно открыть файл: ");
  errMsg += fileName;
  throw errMsg;
```

случае невозможности открыть некоторый файл выглядит следующим образом:

```
}
```

Место программы, в котором исключение *обрабатывается*. При возбуждении исключения нормальное выполнение программы приостанавливается и управление передается *обработчику* исключения. Поиск нужного обработчика часто включает в себя раскрутку так называемого *стека вызовов программы*. После обработки исключения выполнение программы возобновляется, но не с того места, где произошло исключение, а с точки, следующей за обработчиком. Для определения обработчика исключения в C++ используется ключевое слово `catch`. Вот как может выглядеть обработчик для примера из предыдущего абзаца:

```
catch (string exceptionMsg) {
  log_message (exceptionMsg);
  return false;
}
```

Каждый `catch`-обработчик ассоциирован с исключениями, возникающими в блоке операторов, который непосредственно предшествует обработчику и помечен ключевым словом `try`. Одному `try`-блоку могут соответствовать несколько `catch`-предложений,

```
int* stats (const int *ia, int size)
```

каждое из которых относится к определенному виду исключений. Приведем пример:

```
{
  int *pstats = new int [4];
```

```
  try {
    pstats[0] = sum_it (ia,size);
    pstats[1] = min_val (ia,size);
    pstats[2] = max_val (ia,size);
  }
  catch (string exceptionMsg) {
    // код обработчика
  }
  catch (const statsException &statsExcp) {
    // код обработчика
  }

  pstats [3] = pstats[0] / size;
  do_something (pstats);

  return pstats;
}
```

В данном примере в теле функции `stats()` три оператора заключены в `try`-блок, а четыре – нет. Из этих четырех операторов два способны возбудить исключения.

1) `int *pstats = new int [4];`

Выполнение оператора `new` может окончиться неудачей. Стандартная библиотека C++ предусматривает возбуждение исключения `bad_alloc` в случае невозможности выделить нужное количество памяти. Поскольку в примере не предусмотрен обработчик исключения `bad_alloc`, при его возбуждении выполнение программы закончится аварийно.

2) `do_something (pstats);`

Мы не знаем реализации функции `do_something()`. Любая инструкция этой функции, или функции, вызванной из этой функции, или функции, вызванной из функции, вызванной этой функцией, и так далее, потенциально может возбудить исключение. Если в реализации функции `do_something` и вызываемых из нее предусмотрен обработчик такого исключения, то выполнение `stats()` продолжится обычным образом. Если же такого обработчика нет, выполнение программы аварийно завершится.

Необходимо заметить, что, хотя оператор

```
pstats [3] = pstats[0] / size;
```

может привести к делению на ноль, в стандартной библиотеке не предусмотрен такой тип исключения.

Обратимся теперь к инструкциям, объединенным в try-блок. Если в одной из вызываемых в этом блоке функций – `sum_it()`, `min_val()` или `max_val()` –произойдет исключение, управление будет передано на обработчик, следующий за `try`-блоком и перехватывающий именно это исключение. Ни инструкция, возбудившая исключение, ни следующие за ней инструкции в `try`-блоке выполнены не будут. Представим себе, что при вызове функции `sum_it()` возбуждено исключение:

```
throw string ("Ошибка: adump27832");
```

Выполнение функции `sum_it()` прервется, операторы, следующие в `try`-блоке за вызовом этой функции, также не будут выполнены, и `pstats[0]` не будет инициализирована. Вместо этого возбуждается исключительное состояние и исследуются два `catch`-обработчика. В нашем случае выполняется `catch` с параметром типа `string`:

```
catch (string exceptionMsg) {
  // код обработчика
}
```

После выполнения управление будет передано инструкции, следующей за последним catch-обработчиком, относящимся к данному try-блоку. В нашем случае это

```
pstats [3] = pstats[0] / size;
```

(Конечно, обработчик сам может возбуждать исключения, в том числе – того же типа. В такой ситуации будет продолжено выполнение catch-предложений, определенных в программе, вызвавшей функцию stats().)

```
catch (string exceptionMsg) {
```

Вот пример:

```
  // код обработчика
  cerr << "stats(): исключение: "
       << exceptionMsg
       << endl;
  delete [] pstats;
  return 0;
}
```

В таком случае выполнение вернется в функцию, вызвавшую stats(). Будем считать, что разработчик программы предусмотрел проверку возвращаемого функцией stats() значения и корректную реакцию на нулевое значение.

Функция stats() умеет реагировать на два типа исключений: string и statsException. Исключение любого другого типа игнорируется, и управление передается в вызвавшую функцию, а если и в ней не найдется обработчика, – то в функцию более высокого уровня, и так до функции main().При отсутствии обработчика и там, программа аварийно завершится.

Возможно задание специального обработчика, который реагирует на любой тип

```
catch (...) {
  // обрабатывает любое исключение,
  // однако ему недоступен объект, переданный
  // в обработчик в инструкции throw
```

исключения. Синтаксис его таков:

```
}
```

(Детально обработка исключительных ситуаций рассматривается в главах 11 и 19.)

Упражнение 2.18

Какие ошибочные ситуации могут возникнуть во время выполнения следующей функции:

```
int *alloc_and_init (string file_name)
{
  ifstream infile (file_name)
```

```
  int elem_cnt;
  infile >> elem_cnt;
  int *pi = allocate_array(elem_cnt);

  int elem;
  int index=0;
  while (cin >> elem)
    pi[index++] = elem;

  sort_array(pi,elem_cnt);
  register_data(pi);

  return pi;
}
```

Упражнение 2.19

В предыдущем примере вызываемые функции `allocate_array()`, `sort_array()` и `register_data()` могут возбуждать исключения типов `noMem`, `int` и `string` соответственно. Перепишите функцию `alloc_and_init()`, вставив соответствующие блоки `try` и `catch` для обработки этих исключений. Пусть обработчики просто выводят в `cerr` сообщение об ошибке.

Упражнение 2.20

Усовершенствуйте функцию `alloc_and_init()` так, чтобы она сама возбуждала исключение в случае возникновения всех возможных ошибок (это могут быть исключения, относящиеся к вызываемым функциям `allocate_array()`, `sort_array()` и `register_data()` и какими-то еще операторами внутри функции `alloc_and_init()`). Пусть это исключение имеет тип `string` и строка, передаваемая обработчику, содержит описание ошибки.

2.7. Использование пространства имен

Предположим, что мы хотим предоставить в общее пользование наш класс `Array`, разработанный в предыдущих примерах. Однако не мы одни занимались этой проблемой; возможно, кем-то где-то, скажем, в одном из подразделений компании Intel был создан одноименный класс. Из-за того что имена этих классов совпадают, потенциальные пользователи не могут задействовать оба класса одновременно, они должны выбрать один из них. Эта проблема решается добавлением к имени класса некоторой строки, идентифицирующей его разработчиков, скажем,

```
class Cplusplus_Primer_Third_Edition_Array { ... };
```

Конечно, это тоже не гарантирует уникальность имени, но с большой вероятностью избавит пользователя от данной проблемы. Как, однако, неудобно пользоваться столь длинными именами!

Стандарт C++ предлагает для решения проблемы совпадения имен механизм, называемый *пространством имен*. Каждый производитель программного обеспечения может заключить свои классы, функции и другие объекты в свое собственное пространство имен. Вот как выглядит, например, объявление нашего класса `Array`:

```
namespace Cplusplus_Primer_3E {
  template <class elemType> class Array { ... };
}
```

Ключевое слово `namespace` задает пространство имен, определяющее видимость нашего класса и названное в данном случае `Cplusplus_Primer_3E`. Предположим, что у нас есть классы от других разработчиков, помещенные в другие пространства имен:

```
namespace IBM_Canada_Laboratory {
  template <class elemType> class Array { ... };
  class Matrix { ... };
}

namespace Disney_Feature_Animation {
  class Point { ... };
  template <class elemType> class Array { ... };
}
```

По умолчанию в программе видны объекты, объявленные без явного указания пространства имен; они относятся к *глобальному пространству имен*. Для того чтобы обратиться к объекту из другого пространства, нужно использовать его *квалифицированное имя*, которое состоит из идентификатора пространства имен и идентификатора объекта, разделенных оператором разрешения области видимости (`::`).

```
Cplusplus_Primer_3E::Array<string> text;
```

Вот как выглядят обращения к объектам приведенных выше примеров:

```
IBM_Canada_Laboratory::Matrix mat;
Disney_Feature_Animation::Point origin(5000,5000);
```

Для удобства использования можно назначать *псевдонимы* пространствам имен. Псевдоним выбирают коротким и легким для запоминания. Например:

```
// псевдонимы
namespace LIB = IBM_Canada_Laboratory;
namespace DFA = Disney_Feature_Animation;

int main()
{
  LIB::Array<int> ia(1024);
}
```

Псевдонимы употребляются и для того, чтобы скрыть использование пространств имен. Заменив псевдоним, мы можем сменить набор задействованных функций и классов, причем во всем остальном код программы останется таким же. Исправив только одну строчку в приведенном выше примере, мы получим определение уже совсем другого массива:

```
namespace LIB = Cplusplus_Primer_3E;
int main()
{
  LIB::Array<int> ia(1024);
}
```

Конечно, чтобы это стало возможным, необходимо точное совпадение интерфейсов классов и функций, объявленных в этих пространствах имен. Представим, что класс Array из Disney_Feature_Animation не имеет конструктора с одним параметром – размером. Тогда следующий код вызовет ошибку:

```
namespace LIB = Disney_Feature_Animation;

int main()
{
  LIB::Array<int> ia(1024);
}
```

Еще более удобным является способ использования простого, неквалифицированного имени для обращения к объектам, определенным в некотором пространстве имен. Для этого существует директива using:

```
#include "IBM_Canada_Laboratory.h"

using namespace IBM_Canada_Laboratory;

int main()
{
  // IBM_Canada_Laboratory::Matrix
  Matrix mat(4,4);

  // IBM_Canada_Laboratory::Array
  Array<int> ia(1024);

  // ...
}
```

Пространство имен IBM_Canada_Laboratory становится видимым в программе. Можно сделать видимым не все пространство, а отдельные имена внутри него (селективная директива using):

```
#include "IBM_Canada_Laboratory.h"

using namespace IBM_Canada_Laboratory::Matrix;
// видимым становится только Matrix

int main()
{
  // IBM_Canada_Laboratory::Matrix
  Matrix mat(4,4);

  // Ошибка: IBM_Canada_Laboratory::Array невидим
  Array<int> ia(1024);

  // ...
}
```

Как мы уже упоминали, все компоненты стандартной библиотеки C++ объявлены внутри пространства имен std. Поэтому простого включения заголовочного файла недостаточно, чтобы напрямую пользоваться стандартными функциями и классами:

```
#include <string>

// ошибка: string невидим
string current_chapter = "Обзор C++";
```

```
Необходимо использовать директиву using:
#include <string>
using namespace std;

// Ok: видим string
string current_chapter = "Обзор C++";
```

Заметим, однако, что таким образом мы возвращаемся к проблеме “засорения” глобального пространства имен, ради решения которой и был создан механизм

```
#include <string>
```

именованных пространств. Поэтому лучше использовать либо квалифицированное имя:

```
// правильно: квалифицированное имя
std::string current_chapter = "Обзор C++";
либо селективную директиву using:
#include <string>
using namespace std::string;

// Ok: string видим
string current_chapter = "Обзор C++";
```

Мы рекомендуем пользоваться последним способом.

В большинстве примеров этой книги директивы пространств имен были опущены. Это сделано ради сокращения размера кода, а также потому, что большинство примеров были скомпилированы компилятором, не поддерживающим пространства имен – достаточно недавнего нововведения C++. (Детали применения using-объявлений при работе с стандартной библиотекой C++ обсуждаются в разделе 8.6.)

В нижеследующих главах мы создадим еще четыре класса: `String`, `Stack`, `List` и модификацию `Stack`. Все они будут заключены в одно пространство имен – `Cplusplus_Primer_3E`. (Более подробно работа с пространствами имен рассматривается в главе 8.)

Упражнение 2.21

```
namespace Exercize {
```

Дано пространство имен

```
  template <class elemType>
    class Array { ... };

  template <class EType>
    void print (Array< EType > );

  class String { ... }
  template <class ListType>
    class List { ... };
}
```

и текст программы:

```
int main() {
  const int size = 1024;
  Array<String> as (size);
  List<int> il (size);

  // ...

  Array<String> *pas = new Array<String>(as);
  List<int> *pil = new List<int>(il);

  print (*pas);
}
```

Программа не компилируется, поскольку объявления используемых классов заключены в пространство имен `Exercise`. Модифицируйте код программы, используя

(a) квалифицированные имена

(b) селективную директиву `using`

(c) механизм псевдонимов

(d) директиву using

2.8. Стандартный массив – это вектор

Хотя встроенный массив формально и обеспечивает механизм контейнера, он, как мы видели выше, не поддерживает семантику абстракции контейнера. До принятия стандарта C++ для программирования на таком уровне мы должны были либо приобрести нужный класс, либо реализовать его самостоятельно. Теперь же класс массива является частью стандартной библиотеки C++. Только называется он не массив, а вектор.

Разумеется, вектор реализован в виде шаблона класса. Так, мы можем написать

```
vector<int> ivec(10);
vector<string> svec(10);
```

Есть два существенных отличия нашей реализации шаблона класса `Array` от реализации шаблона класса `vector`. Первое отличие состоит в том, что вектор поддерживает как присваивание значений существующим элементам, так и вставку дополнительных элементов, то есть динамически растет во время выполнения, если программист решил воспользоваться этой его возможностью. Второе отличие более радикально и отражает существенное изменение парадигмы проектирования. Вместо того чтобы поддержать большой набор операций-членов, применимых к вектору, таких, как `sort()`, `min()`, `max()`, `find()`и так далее, класс `vector` предоставляет минимальный набор: операции сравнения на равенство и на меньше, `size()` и `empty()`. Более общие операции, перечисленные выше, определены как независимые обобщенные алгоритмы.

Для использования класса `vector` мы должны включить соответствующий заголовочный файл.

```
#include <vector>
```

```
// разные способы создания объектов типа vector
vector<int> vec0; // пустой вектор

const int size = 8;
const int value = 1024;

// вектор размером 8
// каждый элемент инициализируется 0
vector<int> vec1(size);

// вектор размером 8
// каждый элемент инициализируется числом 1024
vector<int> vec2(size,value);

// вектор размером 4
// инициализируется числами из массива ia
int ia[4] = { 0, 1, 1, 2 };
vector<int> vec3(ia,ia+4);

// vec4 - копия vec2
vector<int> vec4(vec2);
```

Так же, как наш класс Array, класс vector поддерживает операцию доступа по индексу.

```
#include <vector>
```

Вот пример перебора всех элементов вектора:

```
extern int getSize();

void mumble()
{
  int size = getSize();
  vector<int> vec(size);

  for (int ix=0; ix<size; ++ix)
    vec[ix] = ix;

  // ...
}
```

Для такого перебора можно также использовать *итераторную пару*. Итератор – это объект класса, поддерживающего абстракцию указательного типа. В шаблоне класса vector определены две функции-члена – begin() и end(), устанавливающие итератор соответственно на первый элемент вектора и на элемент, который следует за последним. Вместе эти две функции задают диапазон элементов вектора. Используя итератор,

```
#include <vector>
```

предыдущий пример можно переписать таким образом:

```
extern int getSize();

void mumble()
{
  int size = getSize();
  vector<int> vec(size);
```

```
  vector<int>::iterator iter = vec.begin();

  for (int ix=0; iter!=vec.end(); ++iter, ++ix)
    *iter = ix;

  // ...
}
```

Определение переменной `iter`

```
vector<int>::iterator iter = vec.begin();
```

инициализирует ее адресом первого элемента вектора `vec`. `iterator` определен с помощью `typedef` в шаблоне класса `vector`, содержащего элементы типа `int`. Операция инкремента

```
++iter
```

перемещает итератор на следующий элемент вектора. Чтобы получить сам элемент, нужно применить операцию разыменования:

```
*iter
```

В стандартной библиотеке C++ имеется поразительно много функций, работающих с классом `vector`, но определенных не как функции-члены класса, а как набор обобщенных алгоритмов. Вот их неполный перечень:

алгоритмы поиска: `find()`, `find_if()`, `search()`, `binary_search()`, `count()`, `count_if()`;

алгоритмы сортировки и упорядочения: `sort()`, `partial_sort()`, `merge()`, `partition()`, `rotate()`, `reverse()`, `random_shuffle()`;

алгоритмы удаления: `unique()`, `remove()`;

численные алгоритмы: `accumulate()`, `partial_sum()`, `inner_product()`, `adjacent_difference()`;

алгоритмы генерации и изменения последовательности: `generate()`, `fill()`, `transform()`, `copy()`, `for_each()`;

алгоритмы сравнения: `equal()`, `min()`, `max()`.

В число параметров этих обобщенных алгоритмов входит итераторная пара, задающая диапазон элементов вектора, к которым применяется алгоритм. Скажем, чтобы упорядочить все элементы некоторого вектора `ivec`, достаточно написать следующее:

```
sort ( ivec.begin(), ivec.end() );
```

Чтобы применить алгоритм `sort()` только к первой половине вектора, мы напишем:

```
sort ( ivec.begin(), ivec.begin() + ivec.size()/2 );
```

Роль итераторной пары может играть и пара указателей на элементы встроенного массива. Пусть, например, нам дан массив:

```
int ia[7] = { 10, 7, 9, 5, 3, 7, 1 };
```

Упорядочить весь массив можно вызовом алгоритма sort():

```
sort ( ia, ia+7 );
```

Так можно упорядочить первые четыре элемента:

```
sort ( ia, ia+4 );
```

Для использования алгоритмов в программу необходимо включить заголовочный файл

```
#include <algorithm>
```

Ниже приведен пример программы, использующей разнообразные алгоритмы в применении к объекту типа vector:

```
#include <vector>
#include <algorithm>
#include <iostream>

int ia[ 10 ] = {
    51, 23, 7, 88, 41, 98, 12, 103, 37, 6
};

int main()
{
    vector< int > vec( ia, ia+10 );
    vector<int>::iterator    it = vec.begin(),    end_it = vec.end();

    cout << "Начальный массив: ";
    for ( ; it != end_it; ++ it ) cout << *it << ' ';
    cout << "\n";

    // сортировка массива
    sort( vec.begin(), vec.end() );

    cout << "упорядоченный массив:   ";
    it = vec.begin(); end_it = vec.end();
    for ( ; it != end_it; ++ it ) cout << *it << ' ';
        cout << "\n\n";

    int search_value;
    cout << "Введите значение для поиска: ";
    cin >> search_value;

    // поиск элемента
    vector<int>::iterator found;
    found = find( vec.begin(), vec.end(), search_value );

    if ( found != vec.end() )
         cout << "значение найдено!\n\n";
    else cout << "значение найдено!\n\n";

    // инвертирование массива
    reverse( vec.begin(), vec.end() );

    cout << "инвертированный массив: ";
    it = vec.begin(); end_it = vec.end();

    for ( ; it != end_it; ++ it ) cout << *it << ' ';
      cout << endl;

}
```

Стандартная библиотека C++ поддерживает и *ассоциативные массивы*. Ассоциативный массив – это массив, элементы которого можно индексировать не только целыми числами, но и значениями любого типа. В терминологии стандартной библиотеки ассоциативный массив называется *отображением* (map). Например, телефонный справочник может быть представлен в виде ассоциативного массива, где индексами

```
#include <map>
```

служат фамилии абонентов, а значениями элементов – телефонные номера:

```
#include <string>
#include "TelephoneNumber.h"
```

```
map<string, telephoneNum> telephone_directory;
```

(Классы векторов, отображений и других контейнеров в подробностях описываются в главе 6. Мы попробуем реализовать систему текстового поиска, используя эти классы. В главе 12 рассмотрены обобщенные алгоритмы, а в Приложении приводятся примеры их использования.)

В данной главе были очень бегло рассмотрены основные аспекты программирования на C++, основы объектно-ориентированного подхода применительно к данному языку и использование стандартной библиотеки. В последующих главах мы разберем эти вопросы более подробно и систематично.

Упражнение 2.22

```
string pals[] = {
  "pooh", "tiger", "piglet", "eeyore", "kanga" };

(a) vector<string> svec1(pals,pals+5);
(b) vector<int>    ivec1(10);
(c) vector<int>    ivec2(10,10);
(d) vector<string> svec2(svec1);
(e) vector<double> dvec;
```

Поясните результаты каждого из следующих определений вектора:

Упражнение 2.23

Напишите две реализации функции `min()`, объявление которой приведено ниже. Функция должна возвращать минимальный элемент массива. Используйте цикл `for` и перебор элементов с помощью

индекса

```
template <class elemType>
```

итератора

elemType min (const vector<elemType> &vec);

Часть II

Основы языка

Код программы и данные, которыми программа манипулирует, записываются в память компьютера в виде последовательности битов. *Бит* – это мельчайший элемент компьютерной памяти, способная хранить либо 0, либо 1. На физическом уровне это соответствует электрическому напряжению, которое, как известно, либо есть , либо нет. Посмотрев на содержимое памяти компьютера, мы увидим что-нибудь вроде:

00011011011100010110010000111011 ...

Очень трудно придать такой последовательности смысл, но иногда нам приходится манипулировать и подобными неструктурированными данными (обычно нужда в этом возникает при программировании драйверов аппаратных устройств). C++ предоставляет набор операций для работы с битовыми данными. (Мы поговорим об этом в главе 4.)

Как правило, на последовательность битов накладывают какую-либо структуру, группируя биты в *байты* и *слова*. Байт содержит 8 бит, а слово – 4 байта, или 32 бита. Однако определение слова может быть разным в разных операционных системах. Сейчас начинается переход к 64-битным системам, а еще недавно были распространены системы с 16-битными словами. Хотя в подавляющем большинстве систем размер байта одинаков, мы все равно будем называть эти величины машинно-зависимыми.

Так выглядит наша последовательность битов, организованная в байты.

Рис 1.

Адресуемая машинная память

Теперь мы можем говорить, например, о байте с адресом 1040 или о слове с адресом 1024 и утверждать, что байт с адресом 1032 не равен байту с адресом 1040.

Однако мы не знаем, что же представляет собой какой-либо байт, какое-либо машинное слово. Как понять смысл тех или иных 8 бит? Для того чтобы однозначно интерпретировать значение этого байта (или слова, или другого набора битов), мы должны знать тип данных, представляемых данным байтом.

C++ предоставляет набор встроенных типов данных: символьный, целый, вещественный – и набор составных и расширенных типов: строки, массивы, комплексные числа. Кроме того, для действий с этими данными имеется базовый набор операций: сравнение, арифметические и другие операции. Есть также операторы переходов, циклов, условные операторы. Эти элементы языка C++ составляют тот набор кирпичиков, из которых можно построить систему любой сложности. Первым шагом в освоении C++ станет изучение перечисленных базовых элементов, чему и посвящена часть II данной книги.

Глава 3 содержит обзор встроенных и расширенных типов, а также механизмов, с помощью которых можно создавать новые типы. В основном это, конечно, механизм классов, представленный в разделе 2.3. В главе 4 рассматриваются выражения, встроенные операции и их приоритеты, преобразования типов. В главе 5 рассказывается об инструкциях языка. И наконец глава 6 представляет стандартную библиотеку C++ и контейнерные типы – вектор и ассоциативный массив.

3. Типы данных C++

В этой главе приводится обзор *встроенных*, или *элементарных*, типов данных языка C++. Она начинается с определения *литералов*, таких, как `3.14159` или `pi`, а затем вводится понятие *переменной*, или *объекта*, который должен принадлежать к одному из типов данных. Оставшаяся часть главы посвящена подробному описанию каждого встроенного типа. Кроме того, приводятся производные типы данных для строк и массивов, предоставляемые стандартной библиотекой C++. Хотя эти типы не являются элементарными, они очень важны для написания настоящих программ на C++, и нам хочется познакомить с ними читателя как можно раньше. Мы будем называть такие типы данных *расширением* базовых типов C++.

3.1. Литералы

В C++ имеется набор встроенных типов данных для представления целых и вещественных чисел, символов, а также тип данных “символьный массив”, который служит для хранения символьных строк. Тип `char` служит для хранения отдельных символов и небольших целых чисел. Он занимает один машинный байт. Типы `short`, `int` и `long` предназначены для представления целых чисел. Эти типы различаются только диапазоном значений, которые могут принимать числа, а конкретные размеры перечисленных типов зависят от реализации. Обычно `short` занимает половину машинного слова, `int` – одно слово, `long` – одно или два слова. В 32-битных системах `int` и `long`, как правило, одного размера.

Типы `float`, `double` и `long double` предназначены для чисел с плавающей точкой и различаются точностью представления (количеством значащих разрядов) и диапазоном. Обычно `float` (одинарная точность) занимает одно машинное слово, `double` (двойная точность) – два, а `long double` (расширенная точность) – три.

`char`, `short`, `int` и `long` вместе составляют *целые типы*, которые, в свою очередь, могут быть *знаковыми* (`signed`) и *беззнаковыми* (`unsigned`). В знаковых типах самый левый бит служит для хранения знака (`0` – плюс, `1` – минус), а оставшиеся биты содержат значение. В беззнаковых типах все биты используются для значения. 8-битовый тип `signed char` может представлять значения от `-128` до `127`, а `unsigned char` – от `0` до `255`.

Когда в программе встречается некоторое число, например `1`, то это число называется *литералом*, или *литеральной константой*. Константой, потому что мы не можем изменить его значение, и литералом, потому что его значение фигурирует в тексте программы. Литерал является неадресуемой величиной: хотя реально он, конечно, хранится в памяти машины, нет никакого способа узнать его адрес. Каждый литерал имеет определенный тип. Так, `0` имеет тип `int`, `3.14159` – тип `double`.

Литералы целых типов можно записать в десятичном, восьмеричном и шестнадцатеричном виде. Вот как выглядит число 20, представленное десятичным, восьмеричным и шестнадцатеричным литералами:

```
20    // десятичный
024   // восьмеричный
0x14  // шестнадцатеричный
```

Если литерал начинается с 0, он трактуется как восьмеричный, если с 0x или 0X, то как шестнадцатеричный. Привычная запись рассматривается как десятичное число.

По умолчанию все целые литералы имеют тип signed int. Можно явно определить целый литерал как имеющий тип long, приписав в конце числа букву L (используется как прописная L, так и строчная l, однако для удобства чтения не следует употреблять строчную: ее легко перепутать с 1). Буква U (или u) в конце определяет литерал как unsigned int, а две буквы – UL или LU – как тип unsigned long. Например:

```
128u  1024UL    1L    8Lu
```

Литералы, представляющие действительные числа, могут быть записаны как с десятичной точкой, так и в научной (экспоненциальной) нотации. По умолчанию они имеют тип double. Для явного указания типа float нужно использовать суффикс F или f, а для long double - L или l, но только в случае записи с десятичной точкой. Например:

```
3.14159F 0/1f    12.345L 0.0
3el      1.0E-3E 2.       1.0L
```

Слова true и false являются литералами типа bool.

Представимые литеральные символьные константы записываются как символы в одинарных кавычках. Например:

```
'a'    '2'    ','    ' ' (пробел)
```

Специальные символы (табуляция, возврат каретки) записываются как escape-последовательности . Определены следующие такие последовательности (они начинаются с символа обратной косой черты):

```
новая строка             \n
горизонтальная табуляция \t
забой                    \b
вертикальная табуляция   \v
возврат каретки          \r
прогон листа             \f
звонок                   \a
обратная косая черта     \\
вопрос                   \?
одиночная кавычка        \'
двойная кавычка          \"
```

escape-последовательность общего вида имеет форму \ooo, где ooo – от одной до трех восьмеричных цифр. Это число является кодом символа. Используя ASCII-код, мы можем написать следующие литералы:

```
\7 (звонок)   \14 (новая строка)
\0 (null)     \062 ('2')
```

Символьный литерал может иметь префикс L (например, L'a'), что означает специальный тип wchar_t – двухбайтовый символьный тип, который применяется для хранения символов национальных алфавитов, если они не могут быть представлены обычным типом char, как, например, китайские или японские буквы.

Строковый литерал – строка символов, заключенная в двойные кавычки. Такой литерал может занимать и несколько строк, в этом случае в конце строки ставится обратная косая черта. Специальные символы могут быть представлены своими escape-последовательностями. Вот примеры строковых литералов:

```
"" (пустая строка)
"a"
"\nCC\toptions\tfile.[cC]\n"
"a multi-line \
string literal signals its \
continuation with a backslash"
```

Фактически строковый литерал представляет собой массив символьных констант, где по соглашению языков C и C++ последним элементом всегда является специальный символ с кодом 0 (\0).

Литерал 'A' задает единственный символ A, а строковый литерал "A" – массив из двух элементов: 'A' и \0 (пустого символа).

Раз существует тип wchar_t, существуют и литералы этого типа, обозначаемые, как и в случае с отдельными символами, префиксом L:

```
L"a wide string literal"
```

Строковый литерал типа wchar_t – это массив символов того же типа, завершенный нулем.

Если в тесте программы идут подряд два или несколько строковых литералов (типа char или wchar_t), компилятор соединяет их в одну строку. Например, следующий текст

```
"two" "some"
```

породит массив из восьми символов – twosome и завершающий нулевой символ.

```
// this is not a good idea
```

Результат конкатенации строк разного типа не определен. Если написать:

```
"two" L"some"
```

то на каком-то компьютере результатом будет некоторая осмысленная строка, а на другом может оказаться нечто совсем иное. Программы, использующие особенности реализации того или иного компилятора или операционной системы, являются *непереносимыми*. Мы крайне не рекомендуем пользоваться такими конструкциями.

Упражнение 3.1

Объясните разницу в определениях следующих литералов:

```
(a) 'a', L'a', "a", L"a"
(b) 10, 10u, 10L, 10uL, 012, 0*C
```

```
(c) 3.14, 3.14f, 3.14L
```

Упражнение 3.2

```
(a) "Who goes with F\144rgus?\014"
(b) 3.14e1L
(c) "two" L"some"
(d) 1024f
(e) 3.14UL
(f) "multiple line
```

Какие ошибки допущены в приведенных ниже примерах?

```
     comment"
```

3.2. Переменные

```
#include <iostream>
```

Представим себе, что мы решаем задачу возведения 2 в степень 10. Пишем:

```
int main() {
    // a first solution
    cout << "2 raised to the power of 10: ";
    cout << 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2;
    cout << endl;
    return 0;
}
```

Задача решена, хотя нам и пришлось неоднократно проверять, действительно ли 10 раз повторяется литерал 2. Мы не ошиблись в написании этой длинной последовательности двоек, и программа выдала правильный результат – 1024.

Но теперь нас попросили возвести 2 в 17 степень, а потом в 23. Чрезвычайно неудобно каждый раз модифицировать текст программы! И, что еще хуже, очень просто ошибиться, написав лишнюю двойку или пропустив ее... А что делать, если нужно напечатать таблицу степеней двойки от 0 до 15? 16 раз повторить две строки, имеющие

```
cout << "2 в степени X\t";
```

общий вид:

```
cout << 2 * ... * 2;
```

где X последовательно увеличивается на 1, а вместо отточия подставляется нужное число литералов?

Да, мы справились с задачей. Заказчик вряд ли будет вникать в детали, удовлетворившись полученным результатом. В реальной жизни такой подход достаточно часто срабатывает, более того, бывает оправдан: задача решена далеко не самым изящным способом, зато в желаемый срок. Искать более красивый и грамотный вариант может оказаться непрактичной тратой времени.

В данном случае “метод грубой силы” дает правильный ответ, но как же неприятно и скучно решать задачу подобным образом! Мы точно знаем, какие шаги нужно сделать, но сами эти шаги просты и однообразны.

Привлечение более сложных механизмов для той же задачи, как правило, значительно увеличивает время подготовительного этапа. Кроме того, чем более сложные механизмы применяются, тем больше вероятность ошибок. Но даже несмотря на неизбежные ошибки и неверные ходы, применение “высоких технологий” может принести выигрыш в скорости разработки, не говоря уже о том, что эти технологии значительно расширяют наши возможности. И – что интересно! – сам процесс решения может стать привлекательным.

Вернемся к нашему примеру и попробуем “технологически усовершенствовать” его реализацию. Мы можем воспользоваться именованным объектом для хранения значения степени, в которую нужно возвести наше число. Кроме того, вместо повторяющейся

```
#include <iostream>

int main()
{
// objects of type int
  int value = 2;
  int pow  = 10;

  cout << value << " в степени "
          << pow << ": \t";

  int res = 1;

  // оператор цикла:
  // повторить вычисление res
  // до тех пор пока cnt не станет больше pow
  for ( int cnt=1; cnt <= pow; ++cnt )
        res = res * value;

  cout << res << endl;
}
```

последовательности литералов применим оператор цикла. Вот как это будет выглядеть:

`value`, `pow`, `res` и `cnt` – это переменные, которые позволяют хранить, модифицировать и извлекать значения. Оператор цикла `for` повторяет строку вычисления результата `pow` раз.

Несомненно, мы создали гораздо более гибкую программу. Однако это все еще не функция. Чтобы получить настоящую функцию, которую можно использовать в любой программе для вычисления степени числа, нужно выделить общую часть вычислений, а

```
int pow( int val, int exp )
```

конкретные значения задать параметрами.

```
{
    for ( int res = 1; exp > 0; --exp )
          res = res * val;
    return res;
}
```

Теперь получить любую степень нужного числа не составит никакого труда. Вот как

```
#include <iostream>
extern int pow(int,int);
```

реализуется последняя наша задача – напечатать таблицу степеней двойки от 0 до 15:

```
int main()
{
    int val = 2;
    int exp = 15;

    cout << "Степени 2\n";
    for ( int cnt=0; cnt <= exp; ++cnt )
          cout << cnt << ": "
               << pow( val, cnt ) << endl;

    return 0;
}
```

Конечно, наша функция `pow()` все еще недостаточно обобщена и недостаточно надежна. Она не может оперировать вещественными числами, неправильно возводит числа в отрицательную степень – всегда возвращает `1`. Результат возведения большого числа в большую степень может не поместиться в переменную типа `int`, и тогда будет возвращено некоторое случайное неправильное значение. Видите, как непросто, оказывается, писать функции, рассчитанные на широкое применение? Гораздо сложнее, чем реализовать конкретный алгоритм, направленный на решение конкретной задачи.

3.2.1. Что такое переменная

Переменная, или *объект* – это именованная область памяти, к которой мы имеем доступ из программы; туда можно помещать значения и затем извлекать их. Каждая переменная C++ имеет определенный тип, который характеризует размер и расположение этой области памяти, диапазон значений, которые она может хранить, и набор операций,

```
int    student_count;
double salary;
bool   on_loan;
strins street_address;
```

применимых к этой переменной. Вот пример определения пяти объектов разных типов:

```
char   delimiter;
```

Переменная, как и литерал, имеет определенный тип и хранит свое значение в некоторой области памяти. *Адресуемость* – вот чего не хватает литералу. С переменной ассоциируются две величины:

- собственно значение, или r-значение (от read value – значение для чтения), которое хранится в этой области памяти и присуще как переменной, так и литералу;
- значение адреса области памяти, ассоциированной с переменной, или l-значение (от location value – значение местоположения) – место, где хранится r-значение; присуще только объекту.

В выражении

```
ch = ch - '0';
```

переменная ch находится и слева и справа от символа операции присваивания. Справа расположено значение для чтения (ch и символьный литерал '0'): ассоциированные с переменной данные считываются из соответствующей области памяти. Слева – значение местоположения: в область памяти, соотнесенную с переменной ch, помещается результат вычитания. В общем случае левый операнд операции присваивания должен быть l-

```
// ошибки компиляции: значения слева не являются l-значениями

// ошибка: литерал - не l-значение
0 = 1;

// ошибка: арифметическое выражение - не l-значение
```

значением. Мы не можем написать следующие выражения:

```
salary + salary * 0.10 = new_salary;
```

Оператор определения переменной выделяет для нее память. Поскольку объект имеет только одну ассоциированную с ним область памяти, такой оператор может встретиться в программе только один раз. Если же переменная, определенная в одном исходном файле,

```
// файл module0.C
```

должна быть использована в другом, появляются проблемы. Например:

```
// определяет объект fileName
string fileName;
// ... присвоить fileName значение

// файл module1.C
// использует объект fileName

// увы, не компилируется:
// fileName не определен в module1.C
ifstream input_file( fileName );
```

C++ требует, чтобы объект был известен до первого обращения к нему. Это вызвано необходимостью гарантировать правильность использования объекта в соответствии с его типом. В нашем примере модуль module1.C вызовет ошибку компиляции, поскольку переменная fileName не определена в нем. Чтобы избежать этой ошибки, мы должны сообщить компилятору об уже определенной переменной fileName. Это делается с

```
// файл module1.C
```

помощью инструкции *объявления* переменной:

```
// использует объект fileName

// fileName объявляется, то есть программа получает
// информацию об этом объекте без вторичного его определения
extern string fileName;
```

```
ifstream input_file( fileName )
```

Объявление переменной сообщает компилятору, что объект с данным именем, имеющий данный тип, определен где-то в программе. Память под переменную при ее объявлении не отводится. (Ключевое слово `extern` рассматривается в разделе 8.2.)

Программа может содержать сколько угодно объявлений одной и той же переменной, но определить ее можно только один раз. Такие объявления удобно помещать в заголовочные файлы, включая их в те модули, которые этого требуют. Так мы можем хранить информацию об объектах в одном месте и обеспечить удобство ее модификации в случае надобности. (Более подробно о заголовочных файлах мы поговорим в разделе 8.2.)

3.2.2. Имя переменной

Имя переменной, или *идентификатор*, может состоять из латинских букв, цифр и символа подчеркивания. Прописные и строчные буквы в именах различаются. Язык C++ не ограничивает длину идентификатора, однако пользоваться слишком длинными именами типа `gosh_this_is_an_impossibly_name_to_type` неудобно.

Некоторые слова являются ключевыми в C++ и не могут быть использованы в качестве идентификаторов; в таблице 3.1 приведен их полный список.

Таблица 3.1. Ключевые слова C++

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

Чтобы текст вашей программы был более понятным, мы рекомендуем придерживаться общепринятых соглашений об именах объектов:

- имя переменной обычно пишется строчными буквами, например `index` (для сравнения: `Index` – это имя типа, а `INDEX` – константа, определенная с помощью директивы препроцессора `#define`);

- идентификатор должен нести какой-либо смысл, поясняя назначение объекта в программе, например: `birth_date` или `salary`;

если такое имя состоит из нескольких слов, как, например, `birth_date`, то принято либо разделять слова символом подчеркивания (`birth_date`), либо писать каждое следующее слово с большой буквы (`birthDate`). Замечено, что программисты, привыкшие к ОбъектноОриентированномуПодходу предпочитают выделять слова заглавными буквами, в то время как те_кто_много_писал_на_C используют символ подчеркивания. Какой из двух способов лучше – вопрос вкуса.

3.2.3. Определение объекта

В самом простом случае оператор определения объекта состоит из спецификатора типа и

```
double salary;
double wage;
int month;
int day;
int year;
```

имени объекта и заканчивается точкой с запятой. Например:

```
unsigned long distance;
```

В одном операторе можно определить несколько объектов одного типа. В этом случае их имена перечисляются через запятую:

```
double salary, wage;
int month,
    day, year;
unsigned long distance;
```

Простое определение переменной не задает ее начального значения. Если объект определен как глобальный, спецификация C++ гарантирует, что он будет инициализирован нулевым значением. Если же переменная локальная либо динамически размещаемая (с помощью оператора `new`), ее начальное значение не определено, то есть она может содержать некоторое случайное значение.

Использование подобных переменных – очень распространенная ошибка, которую к тому же трудно обнаружить. Рекомендуется явно указывать начальное значение объекта, по крайней мере в тех случаях, когда неизвестно, может ли объект инициализировать сам себя. Механизм классов вводит понятие конструктора по умолчанию, который служит для присвоения значений по умолчанию. (Мы уже сказали об этом в разделе 2.3. Разговор о конструкторах по умолчанию будет продолжен немного позже, в разделах 3.11 и 3.15,

```
int main() {
```

где мы будем разбирать классы `string` и `complex` из стандартной библиотеки.)

```
    // неинициализированный локальный объект
    int ival;

    // объект типа string инициализирован
    // конструктором по умолчанию
    string project;
```

```
    // ...
}
```

Начальное значение может быть задано прямо в операторе определения переменной. В C++ допустимы две формы инициализации переменной – явная, с использованием

```
int ival = 1024;
```

оператора присваивания:

```
string project = "Fantasia 2000";
```

```
int ival( 1024 );
```

и неявная, с заданием начального значения в скобках:

```
string project( "Fantasia 2000" );
```

Оба варианта эквивалентны и задают начальные значения для целой переменной `ival` как 1024 и для строки `project` как `"Fantasia 2000"`.

```
double salary = 9999.99, wage = salary + 0.01;
int month = 08;
```

Явную инициализацию можно применять и при определении переменных списком:

```
      day = 07, year = 1955;
```

Переменная становится видимой (и допустимой в программе) сразу после ее определения, поэтому мы могли проинициализировать переменную `wage` суммой только что определенной переменной `salary` с некоторой константой. Таким образом, определение:

```
// корректно, но бессмысленно
int bizarre = bizarre;
```

является синтаксически допустимым, хотя и бессмысленным.

```
// ival получает значение 0, а dval - 0.0
```

Встроенные типы данных имеют специальный синтаксис для задания нулевого значения:

```
int ival = int();
double dval = double();
```

```
// int() применяется к каждому из 10 элементов
```

В следующем определении:

```
vector< int > ivec( 10 );
```

к каждому из десяти элементов вектора применяется инициализация с помощью int(). (Мы уже говорили о классе vector в разделе 2.8. Более подробно об этом см. в разделе 3.10 и главе 6.)

Переменная может быть инициализирована выражением любой сложности, включая

```
#include <cmath>
```

вызовы функций. Например:

```
#include <string>

double price = 109.99, discount = 0.16;
double sale_price( price * discount );
string pet( "wrinkles" );

extern int get_value();
int val =  get_value();

unsigned abs_val = abs( val );
```

abs() – стандартная функция, возвращающая абсолютное значение параметра. get_value() – некоторая пользовательская функция, возвращающая целое значение.

Упражнение 3.3

```
(a) int car = 1024, auto = 2048;
(b) int ival = ival;
(c) int ival( int() );
(d) double salary = wage = 9999.99;
```

Какие из приведенных ниже определений переменных содержат синтаксические ошибки?

```
(e) cin >> int input_value;
```

Упражнение 3.4

Объясните разницу между l-значением и r-значением. Приведите примеры.

Упражнение 3.5

Найдите отличия в использовании переменных name и student в первой и второй

```
(a) extern string name;
    string name( "exercise 3.5a" );

(b) extern vector<string> students;
```

строчках каждого примера:

```
    vector<string> students;
```

Упражнение 3.6

Какие имена объектов недопустимы в C++? Измените их так, чтобы они стали синтаксически правильными:

```
(a) int double = 3.14159; (b) vector< int > _;
(c) string namespase;     (d) string catch-22;

(e) char 1_or_2 = '1';    (f) float Float = 3.14f;
```

Упражнение 3.7

В чем разница между следующими глобальными и локальными определениями

```
string global_class;
int global_int;
```

переменных?

```
int main() {
    int local_int;
    string local_class;

    // ...
}
```

3.3. Указатели

Указатели и динамическое выделение памяти были вкратце представлены в разделе 2.2. Указатель – это объект, содержащий адрес другого объекта и позволяющий косвенно манипулировать этим объектом. Обычно указатели используются для работы с динамически созданными объектами, для построения связанных структур данных, таких, как связанные списки и иерархические деревья, и для передачи в функции больших объектов – массивов и объектов классов – в качестве параметров.

Каждый указатель ассоциируется с некоторым типом данных, причем их внутреннее представление не зависит от внутреннего типа: и размер памяти, занимаемый объектом типа указатель, и диапазон значений у них одинаков5. Разница состоит в том, как компилятор воспринимает адресуемый объект. Указатели на разные типы могут иметь одно и то же значение, но область памяти, где размещаются соответствующие типы, может быть различной:

- указатель на `int`, содержащий значение адреса `1000`, направлен на область памяти 1000-1003 (в 32-битной системе);
- указатель на `double`, содержащий значение адреса `1000`, направлен на область памяти 1000-1007 (в 32-битной системе).

```
int             *ip1, *ip2;
complex<double> *cp;
string          *pstring;
vector<int>     *pvec;
```

Вот несколько примеров:

---

5 На самом деле для указателей на функции это не совсем так: они отличаются от указателей на данные (см. раздел 7.9).

```
double          *dp;
```

Указатель обозначается звездочкой перед именем. В определении переменных списком звездочка должна стоять перед каждым указателем (см. выше: ip1 и ip2). В примере ниже lp – указатель на объект типа long, а lp2 – объект типа long:

```
long *lp, lp2;
```

В следующем случае fp интерпретируется как объект типа float, а fp2 – указатель на него:

```
float fp, *fp2;
```

Оператор разыменования (*) может отделяться пробелами от имени и даже непосредственно примыкать к ключевому слову типа. Поэтому приведенные определения

```
string *ps;
```

синтаксически правильны и совершенно эквивалентны:

```
string* ps;
```

Однако рекомендуется использовать первый вариант написания: второй способен ввести в заблуждение, если добавить к нему определение еще одной переменной через запятую:

//внимание: ps2 не указатель на строку!

```
string* ps, ps2;
```

Можно предположить, что и ps, и ps2 являются указателями, хотя указатель – только первый из них.

Если значение указателя равно 0, значит, он не содержит никакого адреса объекта.

Пусть задана переменная типа int:

```
int ival = 1024;
```

```
//pi инициализирован нулевым адресом
int *pi = 0;

// pi2 инициализирован адресом ival
int *pi2 = &ival;

// правильно: pi и pi2 содержат адрес ival
pi = pi2;

// pi2 содержит нулевой адрес
```

Ниже приводятся примеры определения и использования указателей на int pi и pi2:

```
pi2 = 0;
```

Указателю не может быть присвоена величина, не являющаяся адресом:

```
// ошибка: pi не может принимать значение int
pi = ival
```

Точно так же нельзя присвоить указателю одного типа значение, являющееся адресом

```
double dval;
```

объекта другого типа. Если определены следующие переменные:

```
double *ps = &dval;
```

```
// ошибки компиляции
```

то оба выражения присваивания, приведенные ниже, вызовут ошибку компиляции:

```
// недопустимое присваивание типов данных: int* <== double*
pi = pd
pi = &dval;
```

Дело не в том, что переменная `pi` не может содержать адреса объекта `dval` – адреса объектов разных типов имеют одну и ту же длину. Такие операции смешения адресов запрещены сознательно, потому что интерпретация объектов компилятором зависит от типа указателя на них.

Конечно, бывают случаи, когда нас интересует само значение адреса, а не объект, на который он указывает (допустим, мы хотим сравнить этот адрес с каким-то другим). Для разрешения таких ситуаций введен специальный указатель `void`, который может

```
// правильно: void* может содержать
```

указывать на любой тип данных, и следующие выражения будут правильны:

```
// адреса любого типа
void *pv = pi;
pv = pd;
```

Тип объекта, на который указывает `void*`, неизвестен, и мы не можем манипулировать этим объектом. Все, что мы можем сделать с таким указателем, – присвоить его значение другому указателю или сравнить с какой-либо адресной величиной. (Более подробно мы расскажем об указателе типа `void` в разделе 4.14.)

Для того чтобы обратиться к объекту, имея его адрес, нужно применить операцию *разыменования*, или *косвенную адресацию*, обозначаемую звездочкой (*). Имея

```
int ival = 1024;, ival2 = 2048;
```

следующие определения переменных:

```
int *pi = &ival;
```

мы можем читать и сохранять значение `ival`, применяя операцию разыменования к указателю `pi`:

```
// косвенное присваивание переменной ival значения ival2
*pi = ival2;

// косвенное использование переменной ival как rvalue и lvalue
*pi = abs(*pi); // ival = abs(ival);

*pi = *pi + 1;  // ival = ival + 1;
```

Когда мы применяем операцию взятия адреса (&) к объекту типа int, то получаем результат типа int*

```
int *pi = &ival;
```

Если ту же операцию применить к объекту типа int* (указатель на int), мы получим указатель на указатель на int, т.е. int**. int** – это адрес объекта, который содержит адрес объекта типа int. Разыменовывая ppi, мы получаем объект типа int*, содержащий адрес ival. Чтобы получить сам объект ival, операцию разыменования к

```
int **ppi = &pi;
int *pi2 = *ppi;

cout << "Значение ival\n"
     << "явное значение: " << ival << "\n"
     << "косвенная адресация: " << *pi << "\n"
     << "дважды косвенная адресация: " << **ppi << "\n"
```

ppi необходимо применить дважды.

```
     << endl;
```

Указатели могут быть использованы в арифметических выражениях. Обратите внимание на следующий пример, где два выражения производят совершенно различные действия:

```
int i, j, k;
int *pi = &i;

// i = i + 2
*pi = *pi + 2;

// увеличение адреса, содержащегося в pi, на 2
pi = pi + 2;
```

К указателю можно прибавлять целое значение, можно также вычитать из него. Прибавление к указателю 1 увеличивает содержащееся в нем значение на размер области памяти, отводимой объекту соответствующего типа. Если тип char занимает 1 байт, int – 4 и double – 8, то прибавление 2 к указателям на char, int и double увеличит их значение соответственно на 2, 8 и 16. Как это можно интерпретировать? Если объекты одного типа расположены в памяти друг за другом, то увеличение указателя на 1 приведет к тому, что он будет указывать на следующий объект. Поэтому арифметические действия с указателями чаще всего применяются при обработке массивов; в любых других случаях они вряд ли оправданы.

Вот как выглядит типичный пример использования адресной арифметики при переборе элементов массива с помощью итератора:

```
int ia[10];
```

```
int *iter = &ia[0];
int *iter_end = &ia[10];

while (iter != iter_end) {
  do_something_with_value (*iter);
  ++iter;
}
```

Упражнение 3.8

```
int ival = 1024, ival2 = 2048;
```

Даны определения переменных:

```
int *pi1 = &ival, *pi2 = &ival2, **pi3 = 0;
```

Что происходит при выполнении нижеследующих операций присваивания? Допущены

```
(a) ival = *pi3;   (e) pi1 = *pi3;
(b) *pi2 = *pi3;   (f) ival = *pi1;
(c) ival = pi2;    (g) pi1 = ival;
```

ли в данных примерах ошибки?

```
(d) pi2 = *pi1;    (h) pi3 = &pi2;
```

Упражнение 3.9

Работа с указателями – один из важнейших аспектов C и C++, однако в ней легко

```
pi = &ival;
```

допустить ошибку. Например, код

```
pi = pi + 1024;
```

почти наверняка приведет к тому, что `pi` будет указывать на случайную область памяти. Что делает этот оператор присваивания и в каком случае он не приведет к ошибке?

Упражнение 3.10

Данная программа содержит ошибку, связанную с неправильным использованием

```
int foobar(int *pi) {
  *pi = 1024;
  return *pi;
}

int main() {
  int *pi2 = 0;
  int ival = foobar(pi2);
  return 0;
```

указателей:

```
}
```

В чем состоит ошибка? Как можно ее исправить?

Упражнение 3.11

Ошибки из предыдущих двух упражнений проявляются и приводят к фатальным последствиям из-за отсутствия в C++ проверки правильности значений указателей во время работы программы. Как вы думаете, почему такая проверка не была реализована? Можете ли вы предложить некоторые общие рекомендации для того, чтобы работа с указателями была более безопасной?

3.4. Строковые типы

В C++ поддерживаются два типа строк – встроенный тип, доставшийся от C, и класс `string` из стандартной библиотеки C++. Класс `string` предоставляет гораздо больше возможностей и поэтому удобней в применении, однако на практике нередки ситуации, когда необходимо пользоваться встроенным типом либо хорошо понимать, как он устроен. (Одним из примеров может являться разбор параметров командной строки, передаваемых в функцию `main()`. Мы рассмотрим это в главе 7.)

3.4.1. Встроенный строковый тип

Как уже было сказано, встроенный строковый тип перешел к C++ по наследству от C. Строка символов хранится в памяти как массив, и доступ к ней осуществляется при помощи указателя типа `char*`. Стандартная библиотека C предоставляет набор функций

```
// возвращает длину строки
int strlen( const char* );

// сравнивает две строки
int strcmp( const char*, const char* );

// копирует одну строку в другую
char* strcpy( char*, const char* );
```

для манипулирования строками. Например:

Стандартная библиотека C является частью библиотеки C++. Для ее использования мы должны включить заголовочный файл:

```
#include <cstring>
```

Указатель на `char`, с помощью которого мы обращаемся к строке, указывает на соответствующий строке массив символов. Даже когда мы пишем строковый литерал, например

```
const char *st = "Цена бутылки вина\n";
```

компилятор помещает все символы строки в массив и затем присваивает `st` адрес первого элемента массива. Как можно работать со строкой, используя такой указатель?

Обычно для перебора символов строки применяется адресная арифметика. Поскольку строка всегда заканчивается нулевым символом, можно увеличивать указатель на 1, пока очередным символом не станет нуль. Например:

```
while (*st++ ) { ... }
```

st разыменовывается, и получившееся значение проверяется на истинность. Любое отличное от нуля значение считается истинным, и, следовательно, цикл заканчивается, когда будет достигнут символ с кодом 0. Операция инкремента ++ прибавляет 1 к указателю st и таким образом сдвигает его к следующему символу.

Вот как может выглядеть реализация функции, возвращающей длину строки. Отметим, что, поскольку указатель может содержать нулевое значение (ни на что не указывать),

```
int string_length( const char *st )
{
    int cnt = 0;
    if ( st )
         while ( *st++ )
                  ++cnt;
    return cnt;
}
```

перед операцией разыменования его следует проверять:

Строка встроенного типа может считаться пустой в двух случаях: если указатель на строку имеет нулевое значение (тогда у нас вообще нет никакой строки) или указывает на массив, состоящий из одного нулевого символа (то есть на строку, не содержащую ни одного значимого символа).

```
// pc1 не адресует никакого массива символов
char *pc1 = 0;

// pc2 адресует нулевой символ
const char *pc2 = "";
```

Для начинающего программиста использование строк встроенного типа чревато ошибками из-за слишком низкого уровня реализации и невозможности обойтись без адресной арифметики. Ниже мы покажем некоторые типичные погрешности, допускаемые новичками. Задача проста: вычислить длину строки. Первая версия неверна.

```
#include <iostream>
```

Исправьте ее.

```
const char *st = "Цена бутылки вина\n";

int main() {
    int len = 0;
    while ( st++ ) ++len;

    cout << len << ": " << st;
    return 0;
}
```

В этой версии указатель st не разыменовывается. Следовательно, на равенство 0 проверяется не символ, на который указывает st, а сам указатель. Поскольку изначально этот указатель имел ненулевое значение (адрес строки), то он никогда не станет равным нулю, и цикл будет выполняться бесконечно.

Во второй версии программы эта погрешность устранена. Программа успешно

```
#include <iostream>
```

заканчивается, однако полученный результат неправилен. Где мы не правы на этот раз?

```
const char *st = "Цена бутылки вина\n";

int main()
{
    int len = 0;
    while ( *st++ ) ++len;

    cout << len << ": " << st << endl;
    return 0;
}
```

Ошибка состоит в том, что после завершения цикла указатель `st` адресует не исходный символьный литерал, а символ, расположенный в памяти после завершающего нуля этого литерала. В этом месте может находиться что угодно, и выводом программы будет случайная последовательность символов.

```
st = st – len;
```

Можно попробовать исправить эту ошибку:

```
cout << len << ": " << st;
```

Теперь наша программа выдает что-то осмысленное, но не до конца. Ответ выглядит так:

```
18: ена бутылки вина
```

Мы забыли учесть, что заключительный нулевой символ не был включен в подсчитанную длину. `st` должен быть смещен на длину строки *плюс 1*. Вот, наконец, правильный оператор:

```
st = st – len - 1;
```

а вот и и правильный результат:

```
18: Цена бутылки вина
```

Однако нельзя сказать, что наша программа выглядит элегантно. Оператор

```
st = st – len - 1;
```

добавлен для того, чтобы исправить ошибку, допущенную на раннем этапе проектирования программы, – непосредственное увеличение указателя `st`. Этот оператор не вписывается в логику программы, и код теперь трудно понять. Исправления такого рода часто называют *заплатками* – нечто, призванное заткнуть дыру в существующей программе. Гораздо лучшим решением было бы пересмотреть логику. Одним из

вариантов в нашем случае может быть определение второго указателя, инициализированного значением st:

```
const char *p = st;
```

Теперь p можно использовать в цикле вычисления длины, оставив значение st неизменным:

```
while ( *p++ )
```

3.4.2. Класс string

Как мы только что видели, применение встроенного строкового типа чревато ошибками и не очень удобно из-за того, что он реализован на слишком низком уровне. Поэтому достаточно распространена разработка собственного класса или классов для представления строкового типа – чуть ли не каждая компания, отдел или индивидуальный проект имели свою собственную реализацию строки. Да что говорить, в предыдущих двух изданиях этой книги мы делали то же самое! Это порождало проблемы совместимости и переносимости программ. Реализация стандартного класса string стандартной библиотекой C++ призвана была положить конец этому изобретению велосипедов.

Попробуем специфицировать минимальный набор операций, которыми должен обладать класс string:

- инициализация массивом символов (строкой встроенного типа) или другим объектом типа string. Встроенный тип не обладает второй возможностью;
- копирование одной строки в другую. Для встроенного типа приходится использовать функцию strcpy();
- доступ к отдельным символам строки для чтения и записи. Во встроенном массиве для этого применяется операция взятия индекса или косвенная адресация;
- сравнение двух строк на равенство. Для встроенного типа используется функция strcmp();
- конкатенация двух строк, получая результат либо как третью строку, либо вместо одной из исходных. Для встроенного типа применяется функция strcat(), однако чтобы получить результат в новой строке, необходимо последовательно задействовать функции strcpy() и strcat();
- вычисление длины строки. Узнать длину строки встроенного типа можно с помощью функции strlen();
- возможность узнать, пуста ли строка. У встроенных строк для этой цели

```
char str = 0;
//...
if ( ! str || ! *str )
```

приходится проверять два условия:

```
  return;
```

Класс string стандартной библиотеки C++ реализует все перечисленные операции (и гораздо больше, как мы увидим в главе 6). В данном разделе мы научимся пользоваться основными операциями этого класса.

Для того чтобы использовать объекты класса string, необходимо включить соответствующий заголовочный файл:

```
#include <string>
```

Вот пример строки из предыдущего раздела, представленной объектом типа string и

```
#include <string>
```

инициализированной строкой символов:

```
string st( "Цена бутылки вина\n" );
```

Длину строки возвращает функция-член size() (длина не включает завершающий

```
cout << "Длина "
     << st
     << ": " << st.size()
     << " символов, включая символ новой строки\n";
```

нулевой символ).

Вторая форма определения строки задает пустую строку:

```
string st2; // пустая строка
```

Как мы узнаем, пуста ли строка? Конечно, можно сравнить ее длину с 0:

```
if ( ! st.size() )
     // правильно: пустая
```

Однако есть и специальный метод empty(), возвращающий true для пустой строки и false для непустой:

```
if ( st.empty() )
     // правильно: пустая
```

Третья форма конструктора инициализирует объект типа string другим объектом того же типа:

```
string st3( st );
```

Строка st3 инициализируется строкой st. Как мы можем убедиться, что эти строки совпадают? Воспользуемся оператором сравнения (==):

```
if ( st == st3 )
     // инициализация сработала
```

Как скопировать одну строку в другую? С помощью обычной операции присваивания:

```
st2 = st3; // копируем st3 в st2
```

Для конкатенации строк используется операция сложения (+) или операция сложения с присваиванием (+=). Пусть даны две строки:

```
string s1( "hello, " );
string s2( "world\n" );
```

Мы можем получить третью строку, состоящую из конкатенации первых двух, таким образом:

```
string s3 = s1 + s2;
```

Если же мы хотим добавить s2 в конец s1, мы должны написать:

```
s1 += s2;
```

Операция сложения может конкатенировать объекты класса string не только между собой, но и со строками встроенного типа. Можно переписать пример, приведенный выше, так, чтобы специальные символы и знаки препинания представлялись встроенным типом, а значимые слова – объектами класса string:

```
string s1( "hello" );

const char *pc = ", ";
string s2( "world" );

string s3 = s1 + pc + s2 + "\n";
```

Подобные выражения работают потому, что компилятор знает, как автоматически преобразовывать объекты встроенного типа в объекты класса string. Возможно и простое присваивание встроенной строки объекту string:

```
string s1;
const char *pc = "a character array";

s1 = pc; // правильно
```

Обратное преобразование, однако, не работает. Попытка выполнить следующую инициализацию строки встроенного типа вызовет ошибку компиляции:

```
char *str = s1; // ошибка компиляции
```

Чтобы осуществить такое преобразование, необходимо явно вызвать функцию-член с несколько странным названием c_str():

```
char *str = s1.c_str(); // почти правильно
```

Функция c_str() возвращает указатель на символьный массив, содержащий строку объекта string в том виде, в каком она находилась бы во встроенном строковом типе.

Приведенный выше пример инициализации указателя char *str все еще не совсем корректен. c_str() возвращает указатель на константный массив, чтобы предотвратить возможность непосредственной модификации содержимого объекта через этот указатель, имеющий тип

```
const char *
```

(В следующем разделе мы расскажем о ключевом слове const). Правильный вариант инициализации выглядит так:

```
const char *str = s1.c_str(); // правильно
```

К отдельным символам объекта типа string, как и встроенного типа, можно обращаться с помощью операции взятия индекса. Вот, например, фрагмент кода, заменяющего все

```
string str( "fa.disney.com" );

int size = str.size();
for ( int ix = 0; ix < size; ++ix )
      if ( str[ ix ] == '.' )
```

точки символами подчеркивания:

```
            str[ ix ] = '_';
```

Вот и все, что мы хотели сказать о классе string прямо сейчас. На самом деле, этот класс обладает еще многими интересными свойствами и возможностями. Скажем, предыдущий пример реализуется также вызовом одной-единственной функции replace():

```
replace( str.begin(), str.end(), '.', '_' );
```

replace() – один из обобщенных алгоритмов, с которыми мы познакомились в разделе 2.8 и которые будут детально разобраны в главе 12. Эта функция пробегает диапазон от begin() до end(), которые возвращают указатели на начало и конец строки, и заменяет элементы, равные третьему своему параметру, на четвертый.

Упражнение 3.12

```
(a) char ch = "The long and winding road";
(b) int ival = &ch;
(c) char *pc = &ival;
(d) string st( &ch );

(e) pc = 0;     (i) pc = '0';
(f) st = pc;    (j) st = &ival;
(g) ch = pc[0]; (k) ch = *pc;
```

Найдите ошибки в приведенных ниже операторах:

```
(h) pc = st;    (l) *pc = ival;
```

Упражнение 3.13

Объясните разницу в поведении следующих операторов цикла:

```
while ( st++ )
        ++cnt;

while ( *st++ )
        ++cnt;
```

Упражнение 3.14

Даны две семантически эквивалентные программы. Первая использует встроенный

```
// ***** Реализация с использованием C-строк *****
```

строковый тип, вторая – класс string:

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

int main()
{
    int errors = 0;
    const char *pc = "a very long literal string";

    for ( int ix = 0; ix < 1000000; ++ix )
    {
          int len = strlen( pc );
          char *pc2 = new char[ len + 1 ];
          strcpy( pc2, pc );
          if ( strcmp( pc2, pc ))
               ++errors;

          delete [] pc2;
    }
    cout << "C-строки: "
         << errors << " ошибок.\n";
}
```

```
// ***** Реализация с использованием класса string *****
```

```
#include <iostream>
#include <string>

int main()
{
    int errors = 0;
    string str( "a very long literal string" );

    for ( int ix = 0; ix < 1000000; ++ix )
    {
          int len = str.size();
          string str2 = str;
          if ( str != str2 )
    }
    cout << "класс string: "
         << errors << " ошибок.\n;
```

```
}
```

Что эти программы делают?

Оказывается, вторая реализация выполняется в два раза быстрее первой. Ожидали ли вы такого результата? Как вы его объясните?

Упражнение 3.15

Могли бы вы что-нибудь улучшить или дополнить в наборе операций класса `string`, приведенных в последнем разделе? Поясните свои предложения.

3.5. Спецификатор const

```
for ( int index = 0; index < 512; ++index )
```

Возьмем следующий пример кода:

```
      ... ;
```

С использованием литерала 512 связаны две проблемы. Первая состоит в легкости восприятия текста программы. Почему верхняя граница переменной цикла должна быть равна именно 512? Что скрывается за этой величиной? Она кажется случайной...

Вторая проблема касается простоты модификации и сопровождения кода. Предположим, программа состоит из 10 000 строк, и литерал 512 встречается в 4% из них. Допустим, в 80% случаев число 512 должно быть изменено на 1024. Способны ли вы представить трудоемкость такой работы и количество ошибок, которые можно сделать, исправив не то значение?

Обе эти проблемы решаются одновременно: нужно создать объект со значением 512. Присвоив ему осмысленное имя, например `bufSize`, мы сделаем программу гораздо более понятной: ясно, с чем именно сравнивается переменная цикла.

```
index < bufSize
```

В этом случае изменение размера `bufSize` не требует просмотра 400 строк кода для модификации 320 из них. Насколько уменьшается вероятность ошибок ценой добавления

```
int bufSize = 512; // размер буфера ввода
// ...
```

всего одного объекта! Теперь значение 512 *локализовано*.

```
for ( int index = 0; index < bufSize; ++index )
      // ...
```

Остается одна маленькая проблема: переменная `bufSize` здесь является l-значением, которое можно случайно изменить в программе, что приведет к трудно отлавливаемой ошибке. Вот одна из распространенных ошибок – использование операции присваивания (=) вместо сравнения (==):

```
// случайное изменение значения bufSize
```

```
if ( bufSize = 1 )
     // ...
```

В результате выполнения этого кода значение bufSize станет равным 1, что может привести к совершенно непредсказуемому поведению программы. Ошибки такого рода обычно очень тяжело обнаружить, поскольку они попросту не видны.

Использование спецификатора const решает данную проблему. Объявив объект как

```
const int bufSize = 512; // размер буфера ввода
```

мы превращаем переменную в константу со значением 512, значение которой не может быть изменено: такие попытки пресекаются компилятором: неверное использование оператора присваивания вместо сравнения, как в приведенном примере, вызовет ошибку

```
// ошибка: попытка присваивания значения константе
```

компиляции.

```
if ( bufSize = 0 ) ...
```

Раз константе нельзя присвоить значение, она должна быть инициализирована в месте своего определения. Определение константы без ее инициализации также вызывает ошибку компиляции:

```
const double pi; // ошибка: неинициализированная константа
```

Давайте рассуждать дальше. Явная трансформация значения константы пресекается компилятором. Но как быть с косвенной адресацией? Можно ли присвоить адрес

```
const double minWage = 9.60;

// правильно? ошибка?
```

константы некоторому указателю?

```
double *ptr = &minWage;
```

Должен ли компилятор разрешить подобное присваивание? Поскольку minWage – константа, ей нельзя присвоить значение. С другой стороны, ничто не запрещает нам написать:

```
*ptr += 1.40; // изменение объекта minWage!
```

Как правило, компилятор не в состоянии уберечь от использования указателей и не сможет сигнализировать об ошибке в случае подобного их употребления. Для этого требуется слишком глубокий анализ логики программы. Поэтому компилятор просто запрещает присваивание адресов констант обычным указателям.

Что же, мы лишены возможности использовать указатели на константы? Нет. Для этого существуют указатели, объявленные со спецификатором const:

```
const double *cptr;
```

где cptr – указатель на объект типа const double. Тонкость заключается в том, что сам

```
const double *pc = 0;
const double minWage = 9.60;

// правильно: не можем изменять minWage с помощью pc
pc = &minWage;

double dval = 3.14;

// правильно: не можем изменять minWage с помощью pc
// хотя dval и не константа
pc = &dval; // правильно

dval = 3.14159; //правильно
```

указатель – не константа, а значит, мы можем изменять его значение. Например:

```
*pc = 3.14159; // ошибка
```

Адрес константного объекта присваивается только указателю на константу. Вместе с тем, такому указателю может быть присвоен и адрес обычной переменной:

```
pc = &dval;
```

Константный указатель не позволяет изменять адресуемый им объект с помощью косвенной адресации. Хотя dval в примере выше и не является константой, компилятор не допустит изменения переменной dval через pc. (Опять-таки потому, что он не в состоянии определить, адрес какого объекта может содержать указатель в произвольный момент выполнения программы.)

В реальных программах указатели на константы чаще всего употребляются как формальные параметры функций. Их использование дает гарантию, что объект, переданный в функцию в качестве фактического аргумента, не будет изменен этой

```
// В реальных программах указатели на константы чаще всего
// употребляются как формальные параметры функций
```

функцией. Например:

```
int strcmp( const char *str1, const char *str2 );
```

(Мы еще поговорим об указателях на константы в главе 7, когда речь пойдет о функциях.)

Существуют и константные указатели. (Обратите внимание на разницу между константным указателем и указателем на константу!). Константный указатель может адресовать как константу, так и переменную. Например:

```
int errNumb = 0;
int *const currErr = &errNumb;
```

Здесь curErr – константный указатель на неконстантный объект. Это значит, что мы не можем присвоить ему адрес другого объекта, хотя сам объект допускает модификацию. Вот как мог бы быть использован указатель curErr:

```
do_something();
```

```
      errorHandler();
      *curErr = 0; // правильно: обнулим значение errNumb
if ( *curErr ) {
}
```

Попытка присвоить значение константному указателю вызовет ошибку компиляции:

```
curErr = &myErNumb; // ошибка
```

Константный указатель на константу является объединением двух рассмотренных

```
const double pi = 3.14159;
```

случаев.

```
const double *const pi_ptr = &pi;
```

Ни значение объекта, на который указывает `pi_ptr`, ни значение самого указателя не может быть изменено в программе.

Упражнение 3.16

```
(a) int i;          (d) int *const cpi;
(b) const int ic;   (e) const int *const cpic;
```

Объясните значение следующих пяти определений. Есть ли среди них ошибочные?

```
(c) const int *pic;
```

Упражнение 3.17

```
(a) int i = -1;
(b) const int ic = i;
(c) const int *pic = &ic;
(d) int *const cpi = &ic;
```

Какие из приведенных определений правильны? Почему?

```
(e) const int *const cpic = &ic;
```

Упражнение 3.18

Используя определения из предыдущего упражнения, укажите правильные операторы

```
(a) i = ic;     (d) pic = cpic;
(b) pic = &ic;  (i) cpic = &ic;
```

присваивания. Объясните.

```
(c) cpi = pic;  (f) ic = *cpic;
```

3.6. Ссылочный тип

Ссылочный тип, иногда называемый псевдонимом, служит для задания объекту дополнительного имени. Ссылка позволяет косвенно манипулировать объектом, точно так же, как это делается с помощью указателя. Однако эта косвенная манипуляция не требует специального синтаксиса, необходимого для указателей. Обычно ссылки употребляются как формальные параметры функций. В этом разделе мы рассмотрим самостоятельное использование объектов ссылочного типа.

Ссылочный тип обозначается указанием оператора взятия адреса (&) перед именем

```
int ival = 1024;

// правильно: refVal - ссылка на ival
int &refVal = ival;

// ошибка: ссылка должна быть инициализирована
```

переменной. Ссылка должна быть инициализирована. Например:

```
int &refVal2;
```

Хотя, как мы говорили, ссылка очень похожа на указатель, она должна быть инициализирована не адресом объекта, а его значением. Таким объектом может быть и

```
int ival = 1024;

// ошибка: refVal имеет тип int, а не int*
int &refVal = &ival;

int *pi = &ival;

// правильно: ptrVal - ссылка на указатель
```

указатель:

```
int *&ptrVal2 = pi;
```

Определив ссылку, вы уже не сможете изменить ее так, чтобы работать с другим объектом (именно поэтому ссылка должна быть инициализирована в месте своего определения). В следующем примере оператор присваивания не меняет значения refVal,

```
int min_val = 0;

// ival получает значение min_val,
// а не refVal меняет значение на min_val
```

новое значение присваивается переменной ival – ту, которую адресует refVal.

```
refVal = min_val;
```

Все операции со ссылками реально воздействуют на адресуемые ими объекты. В том числе и операция взятия адреса. Например:

```
refVal += 2;
```

прибавляет 2 к ival – переменной, на которую ссылается refVal. Аналогично

```
int ii = refVal;
```

присваивает ii текущее значение ival,

```
int *pi = &refVal;
```

инициализирует pi адресом ival.

Если мы определяем ссылки в одной инструкции через запятую, перед каждым объектом типа ссылки должен стоять амперсанд (&) – оператор взятия адреса (точно так же, как и

```
// определено два объекта типа int
int ival = 1024, ival2 = 2048;

// определена одна ссылка и один объект
int &rval = ival, rval2 = ival2;

// определен один объект, один указатель и одна ссылка
int inal3 = 1024, *pi = ival3, &ri = ival3;

// определены две ссылки
```

для указателей). Например:

```
int &rval3 = ival3, &rval4 = ival2;
```

Константная ссылка может быть инициализирована объектом другого типа (если, конечно, существует возможность преобразования одного типа в другой), а также

```
double dval = 3.14159;

// верно только для константных ссылок
const int &ir = 1024;
const int &ir2 = dval;
```

безадресной величиной – такой, как литеральная константа. Например:

```
const double &dr = dval + 1.0;
```

Если бы мы не указали спецификатор const, все три определения ссылок вызвали бы ошибку компиляции. Однако, причина, по которой компилятор не пропускает таких определений, неясна. Попробуем разобраться.

Для литералов это более или менее понятно: у нас не должно быть возможности косвенно поменять значение литерала, используя указатели или ссылки. Что касается объектов другого типа, то компилятор преобразует исходный объект в некоторый

```
double dval = 1024;
```

вспомогательный. Например, если мы пишем:

```
const int &ri = dval;
```

то компилятор преобразует это примерно так:

```
int temp = dval;
const int &ri = temp;
```

Если бы мы могли присвоить новое значение ссылке `ri`, мы бы реально изменили не `dval`, а `temp`. Значение `dval` осталось бы тем же, что совершенно неочевидно для программиста. Поэтому компилятор запрещает такие действия, и единственная возможность проинициализировать ссылку объектом другого типа – объявить ее как `const`.

Вот еще один пример ссылки, который трудно понять с первого раза. Мы хотим определить ссылку на адрес константного объекта, но наш первый вариант вызывает

```
const int ival = 1024;
// ошибка: нужна константная ссылка
```

ошибку компиляции:

```
int *&pi_ref = &ival;
```

```
const int ival = 1024;
// все равно ошибка
```

Попытка исправить дело добавлением спецификатора `const` тоже не проходит:

```
const int *&pi_ref = &ival;
```

В чем причина? Внимательно прочитав определение, мы увидим, что `pi_ref` является ссылкой на константный указатель на объект типа `int`. А нам нужен неконстантный

```
const int ival = 1024;
// правильно
```

указатель на константный объект, поэтому правильной будет следующая запись:

```
int *const &piref = &ival;
```

Между ссылкой и указателем существуют два основных отличия. Во-первых, ссылка обязательно должна быть инициализирована в месте своего определения. Во-вторых, всякое изменение ссылки преобразует не ее, а тот объект, на который она ссылается. Рассмотрим на примерах. Если мы пишем:

```
int *pi = 0;
```

мы инициализируем указатель `pi` нулевым значением, а это значит, что `pi` не указывает ни на какой объект. В то же время запись

```
const int &ri = 0;
```

означает примерно следующее:

```
int temp = 0;
const int &ri = temp;
```

```
int ival = 1024, ival2 = 2048;
int *pi = &ival, *pi2 = &ival2;
```

Что касается операции присваивания, то в следующем примере:

```
pi = pi2;
```

переменная ival, на которую указывает pi, остается неизменной, а pi получает значение адреса переменной ival2. И pi, и pi2 и теперь указывают на один и тот же объект ival2.

```
int &ri = ival, &ri2 = ival2;
```

Если же мы работаем со ссылками:

```
ri = ri2;
```

то само значение ival меняется, но ссылка ri по-прежнему адресует ival.

В реальных C++ программах ссылки редко используются как самостоятельные объекты,

```
// пример использования ссылок

// Значение возвращается в параметре next_value
bool get_next_value( int &next_value );

// перегруженный оператор
```

обычно они употребляются в качестве формальных параметров функций. Например:

```
Matrix operator+( const Matrix&, const Matrix& );
```

```
int ival;
```

Как соотносятся самостоятельные объекты-ссылки и ссылки-параметры? Если мы пишем:

```
while (get_next_value( ival )) ...
```

это равносильно следующему определению ссылки внутри функции:

```
int &next_value = ival;
```

(Подробнее использование ссылок в качестве формальных параметров функций рассматривается в главе 7.)

Упражнение 3.19

```
(a) int ival = 1.01;       (b) int &rval1 = 1.01;
(c) int &rval2 = ival;     (d) int &rval3 = &ival;
(e) int *pi = &ival;       (f) int &rval4 = pi;
(g) int &rval5 = pi*;      (h) int &*prval1 = pi;
```

Есть ли ошибки в данных определениях? Поясните. Как бы вы их исправили?

```
(i) const int &ival2 = 1;  (j) const int &*prval2 = &ival;
```

Упражнение 3.20

Если ли среди нижеследующих операций присваивания ошибочные (используются

```
(a) rval1 = 3.14159;
(b) prval1 = prval2;
(c) prval2 = rval1;
```

определения из предыдущего упражнения)?

```
(d) *prval2 = ival2;
```

Упражнение 3.21

```
(a) int ival = 0;
    const int *pi = 0;
    const int &ri = 0;

(b) pi = &ival;
    ri = &ival;
```

Найдите ошибки в приведенных инструкциях:

```
    pi = &rval;
```

3.7. Тип bool

```
// инициализация строки
string search_word = get_word();

// инициализация переменной found
bool found = false;

string next_word;
while ( cin >> next_word )
   if ( next_word == search_word )
        found = true;
// ...

// сокращенная запись: if ( found == true )
if ( found )
      cout << "ok, мы нашли слово\n";
```

Объект типа `bool` может принимать одно из двух значений: `true` и `false`. Например:

```
 else cout << "нет, наше слово не встретилось.\n";
```

Хотя `bool` относится к одному из целых типов, он не может быть объявлен как `signed`, `unsigned`, `short` или `long`, поэтому приведенное определение ошибочно:

```
// ошибка
short bool found = false;
```

Объекты типа bool неявно преобразуются в тип int. Значение true превращается в 1, а

```
bool found = false;
int occurrence_count = 0;

while ( /* mumble */ )
{
   found = look_for( /* something */ );

   // значение found преобразуется в 0 или 1
   occurrence_count += found;
}
```

false – в 0. Например:

Таким же образом значения целых типов и указателей могут быть преобразованы в

```
// возвращает количество вхождений
extern int find( const string& );
bool found = false;
if ( found = find( "rosebud" ))
     // правильно: found == true

// возвращает указатель на элемент
extern int* find( int value );

if ( found = find( 1024 ))
     // правильно: found == true
```

значения типа bool. При этом 0 интерпретируется как false, а все остальное как true:

3.8. Перечисления

Нередко приходится определять переменную, которая принимает значения из некоего набора. Скажем, файл открывают в любом из трех режимов: для чтения, для записи, для добавления.

```
const int input = 1;
const int output = 2;
```

Конечно, можно определить три константы для обозначения этих режимов:

```
const int append = 3;
```

и пользоваться этими константами:

```
bool open_file( string file_name, int open_mode);
// ...
open_file( "Phoenix_and_the_Crane", append );
```

Подобное решение допустимо, но не вполне приемлемо, поскольку мы не можем гарантировать, что аргумент, передаваемый в функцию open_file() равен только 1, 2 или 3.

Использование перечислимого типа решает данную проблему. Когда мы пишем:

enum open_modes{ input = 1, output, append };

мы определяем новый тип open_modes. Допустимые значения для объекта этого типа ограничены набором 1, 2 и 3, причем каждое из указанных значений имеет мнемоническое имя. Мы можем использовать имя этого нового типа для определения как объекта данного типа, так и типа формальных параметров функции:

```
void open_file( string file_name, open_modes om );
```

input, output и append являются *элементами перечисления*. Набор элементов перечисления задает допустимое множество значений для объекта данного типа. Переменная типа open_modes (в нашем примере) инициализируется одним из этих значений, ей также может быть присвоено любое из них. Например:

```
open_file( "Phoenix and the Crane", append );
```

Попытка присвоить переменной данного типа значение, отличное от одного из элементов перечисления (или передать его параметром в функцию), вызовет ошибку компиляции. Даже если попробовать передать целое значение, соответствующее одному из элементов

```
// ошибка: 1 не является элементом перечисления open_modes
```

перечисления, мы все равно получим ошибку:

```
open_file( "Jonah", 1 );
```

Есть способ определить переменную типа open_modes, присвоить ей значение одного из

```
open_modes om = input;
// ...
```

элементов перечисления и передать параметром в функцию:

```
om = append;
open_file( "TailTell", om );
```

Однако получить имена таких элементов невозможно. Если мы напишем оператор вывода:

```
cout << input << " " << om << endl;
```

то все равно получим:

```
1 3
```

Эта проблема решается, если определить строковый массив, в котором элемент с индексом, равным значению элемента перечисления, будет содержать его имя. Имея

```
cout << open_modes_table[ input ] << " "
```

такой массив, мы сможем написать:

```
          << open_modes_table[ om ]    << endl
```

Будет выведено:

```
input append
```

```
// не поддерживается
for ( open_modes iter = input; iter != append; ++inter )
```

Кроме того, нельзя перебрать все значения перечисления:

```
// ...
```

Для определения перечисления служит ключевое слово enum, а имена элементов задаются в фигурных скобках, через запятую. По умолчанию первый из них равен 0, следующий – 1 и так далее. С помощью оператора присваивания это правило можно изменить. При этом каждый следующий элемент без явно указанного значения будет на 1 больше, чем элемент, идущий перед ним в списке. В нашем примере мы явно указали значение 1 для

```
// shape == 0, sphere == 1, cylinder == 2, polygon == 3
```

input, при этом output и append будут равны 2 и 3. Вот еще один пример:

```
enum Forms{ share, spere, cylinder, polygon };
```

Целые значения, соответствующие разным элементам одного перечисления, не обязаны

```
// point2d == 2, point2w == 3, point3d == 3, point3w == 4
```

отличаться. Например:

```
enum Points { point2d=2, point2w, point3d=3, point3w=4 };
```

Объект, тип которого – перечисление, можно определять, использовать в выражениях и передавать в функцию как аргумент. Подобный объект инициализируется только значением одного из элементов перечисления, и только такое значение ему присваивается – явно или как значение другого объекта того же типа. Даже соответствующие допустимым элементам перечисления целые значения не могут быть ему присвоены:

```
void mumble() {
    Points pt3d = point3d; // правильно: pt2d == 3

    // ошибка: pt3w инициализируется типом int
    Points pt3w = 3;

    // ошибка: polygon не входит в перечисление Points
    pt3w = polygon;

    // правильно: оба объекта типа Points
    pt3w = pt3d;
}
```

Однако в арифметических выражениях перечисление может быть автоматически

```
const int array_size = 1024;

// правильно: pt2w преобразуется int
```

преобразовано в тип int. Например:

```
int chunk_size = array_size * pt2w;
```

3.9. Тип “массив”

Мы уже касались массивов в разделе 2.1. Массив – это набор элементов одного типа, доступ к которым производится по индексу – порядковому номеру элемента в массиве. Например:

```
int ival;
```

определяет ival как переменную типа int, а инструкция

```
int ia[ 10 ];
```

задает массив из десяти объектов типа int. К каждому из этих объектов, или *элементов массива*, можно обратиться с помощью операции взятия индекса:

```
ival = ia[ 2 ];
```

присваивает переменной ival значение элемента массива ia с индексом 2. Аналогично

```
ia[ 7 ] = ival;
```

присваивает элементу с индексом 7 значение ival.

Определение массива состоит из спецификатора типа, имени массива и размера. Размер задает количество элементов массива (не менее 1) и заключается в квадратные скобки. Размер массива нужно знать уже на этапе компиляции, а следовательно, он должен быть константным выражением, хотя не обязательно задается литералом. Вот примеры правильных и неправильных определений массивов:

```
extern int get_size();

// buf_size и max_files константы
const int buf_size = 512, max_files = 20;
int staff_size = 27;

// правильно: константа
char input_buffer[ buf_size ];

// правильно: константное выражение: 20 - 3
char *fileTable[ max_files-3 ];

// ошибка: не константа
double salaries[ staff_size ];

// ошибка: не константное выражение
int test_scores[ get_size() ];
```

Объекты `buf_size` и `max_files` являются константами, поэтому определения массивов `input_buffer` и `fileTable` правильны. А вот `staff_size` – переменная (хотя и инициализированная константой 27), значит, `salaries[staff_size]` недопустимо. (Компилятор не в состоянии найти значение переменной `staff_size` в момент определения массива `salaries`.)

Выражение `max_files-3` может быть вычислено на этапе компиляции, следовательно, определение массива `fileTable[max_files-3]` синтаксически правильно.

Нумерация элементов начинается с 0, поэтому для массива из 10 элементов правильным диапазоном индексов является не 1 – 10, а 0 – 9. Вот пример перебора всех элементов массива:

```
int main()
{
          const int array_size = 10;
          int ia[ array_size ];

          for ( int ix = 0; ix < array_size; ++ ix )
          ia[ ix ] = ix;
}
```

При определении массив можно явно инициализировать, перечислив значения его элементов в фигурных скобках, через запятую:

```
const int array_size = 3;
int ia[ array_size ] = { 0, 1, 2 };
```

Если мы явно указываем список значений, то можем не указывать размер массива:

```
// массив размера 3
int ia[] = { 0, 1, 2 };
```

компилятор сам подсчитает количество элементов:

Когда явно указаны и размер, и список значений, возможны три варианта. При совпадении размера и количества значений все очевидно. Если список значений короче, чем заданный размер, оставшиеся элементы массива инициализируются нулями. Если же

```
// ia ==> { 0, 1, 2, 0, 0 }
const int array_size = 5;
```

в списке больше значений, компилятор выводит сообщение об ошибке:

```
int ia[ array_size ] = { 0, 1, 2 };
```

Символьный массив может быть инициализирован не только списком символьных значений в фигурных скобках, но и строковым литералом. Однако между этими

```
const char cal[] = {'C', '+', '+' };
```

способами есть некоторая разница. Допустим,

```
const char cal2[] = "C++";
```

Размерность массива `ca1` равна 3, массива `ca2` – 4 (в строковых литералах учитывается

```
// ошибка: строка "Daniel" состоит из 7 элементов
```

завершающий нулевой символ). Следующее определение вызовет ошибку компиляции:

```
const char ch3[ 6 ] = "Daniel";
```

Массиву не может быть присвоено значение другого массива, недопустима и инициализация одного массива другим. Кроме того, не разрешается использовать массив

```
const int array_size = 3;
int ix, jx, kx;

// правильно: массив указателей типа int*
int *iar [] = { &ix, &jx, &kx };

// error: массивы ссылок недопустимы
int &iar[] = { ix, jx, kx };

int main()
{
          int ia3{ array_size ]; // правильно

          // ошибка: встроенные массивы нельзя копировать
ia3 = ia;
return 0;
```

ссылок. Вот примеры правильного и неправильного употребления массивов:

```
}
```

Чтобы скопировать один массив в другой, придется проделать это для каждого элемента по отдельности:

```
const int array_size = 7;
int ia1[] = { 0, 1, 2, 3, 4, 5, 6 };

int main()
{
          int ia3[ array_size ];

          for ( int ix = 0; ix < array_size; ++ix )
           ia2[ ix ] = ia1[ ix ];

          return 0;
}
```

В качестве индекса массива может выступать любое выражение, дающее результат целого

```
int someVal, get_index();
```

типа. Например:

```
ia2[ get_index() ] = someVal;
```

Подчеркнем, что язык C++ не обеспечивает контроля индексов массива – ни на этапе компиляции, ни на этапе выполнения. Программист сам должен следить за тем, чтобы индекс не вышел за границы массива. Ошибки при работе с индексом достаточно распространены. К сожалению, не так уж трудно встретить примеры программ, которые компилируются и даже работают, но тем не менее содержат фатальные ошибки, рано или поздно приводящие к краху.

Упражнение 3.22

```
(a) int ia[ buf_size ];     (d) int ia[ 2 * 7 - 14 ]
(b) int ia[ get_size() ];   (e) char st[ 11 ] = "fundamental";
```

Какие из приведенных определений массивов содержат ошибки? Поясните.

```
(c) int ia[ 4 * 7 - 14 ];
```

Упражнение 3.23

Следующий фрагмент кода должен инициализировать каждый элемент массива

```
int main() {
    const int array_size = 10;
    int ia[ array_size ];

    for ( int ix = 1; ix <= array_size; ++ix )
               ia[ ia ] = ix;

    // ...
```

значением индекса. Найдите допущенные ошибки:

```
}
```

3.9.1. Многомерные массивы

В C++ есть возможность использовать многомерные массивы, при объявлении которых необходимо указать правую границу каждого измерения в отдельных квадратных скобках. Вот определение двумерного массива:

```
int ia[ 4 ][ 3 ];
```

Первая величина (4) задает количество строк, вторая (3) – количество столбцов. Объект `ia` определен как массив из четырех строк по три элемента в каждой. Многомерные

```
int ia[ 4 ][ 3 ] = {
    { 0, 1, 2 },
    { 3, 4, 5 },
    { 6, 7, 8 },
    { 9, 10, 11 }
```

массивы тоже могут быть инициализированы:

```
};
```

Внутренние фигурные скобки, разбивающие список значений на строки, необязательны и используются, как правило, для удобства чтения кода. Приведенная ниже инициализация в точности соответствует предыдущему примеру, хотя менее понятна:

```
int ia[4][3] = { 0,1,2,3,4,5,6,7,8,9,10,11 };
```

Следующее определение инициализирует только первые элементы каждой строки. Оставшиеся элементы будут равны нулю:

```
int ia[ 4 ][ 3 ] = { {0}, {3}, {6}, {9} };
```

Если же опустить внутренние фигурные скобки, результат окажется совершенно иным. Все три элемента первой строки и первый элемент второй получат указанное значение, а остальные будут неявно инициализированы 0.

```
int ia[ 4 ][ 3 ] = { 0, 3, 6, 9 };
```

При обращении к элементам многомерного массива необходимо использовать индексы для каждого измерения (они заключаются в квадратные скобки). Так выглядит

```
int main()
{
          const int rowSize = 4;
          const int colSize = 3;
int ia[ rowSize ][ colSize ];

          for ( int = 0; i < rowSize; ++i )
          for ( int j = 0; j < colSize; ++j )
                ia[ i ][ j ] = i + j j;
```

инициализация двумерного массива с помощью вложенных циклов:

```
}
```

Конструкция

```
ia[ 1, 2 ]
```

является допустимой с точки зрения синтаксиса C++, однако означает совсем не то, чего ждет неопытный программист. Это отнюдь не объявление двумерного массива 1 на 2. Агрегат в квадратных скобках – это список выражений через запятую, результатом которого будет последнее значение 2 (см. оператор "запятая" в разделе 4.2). Поэтому объявление `ia[1,2]` эквивалентно `ia[2]`. Это еще одна возможность допустить ошибку.

3.9.2. Взаимосвязь массивов и указателей

Если мы имеем определение массива:

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
```

то что означает простое указание его имени в программе?

```
ia;
```

Использование идентификатора массива в программе эквивалентно указанию адреса его

```
ia;
```

первого элемента:

```
&ia[0]
```

```
// оба выражения возвращают первый элемент
*ia;
```

Аналогично обратиться к значению первого элемента массива можно двумя способами:

```
ia[0];
```

Чтобы взять адрес второго элемента массива, мы должны написать:

```
&ia[1];
```

Как мы уже упоминали раньше, выражение

```
ia+1;
```

также дает адрес второго элемента массива. Соответственно, его значение дают нам

```
*(ia+1);
```

следующие два способа:

```
ia[1];
```

Отметим разницу в выражениях:

```
*ia+1
```

и

```
*(ia+1);
```

Операция разыменования имеет более высокий приоритет, чем операция сложения (о приоритетах операций говорится в разделе 4.13). Поэтому первое выражение сначала разыменовывает переменную ia и получает первый элемент массива, а затем прибавляет к нему 1. Второе же выражение доставляет значение второго элемента.

Проход по массиву можно осуществлять с помощью индекса, как мы делали это в

```
#include <iostream>
int main()
{
          int ia[9] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
          int *pbegin = ia;
```

предыдущем разделе, или с помощью указателей. Например:

```
     while ( pbegin != pend ) {
       cout << *pbegin <<;
       ++pbegin;
```

```
          int *pend = ia + 9;
}
```

Указатель pbegin инициализируется адресом первого элемента массива. Каждый проход по циклу увеличивает этот указатель на 1, что означает смещение его на следующий элемент. Как понять, где остановиться? В нашем примере мы определили второй указатель pend и инициализировали его адресом, следующим за последним элементом массива ia. Как только значение pbegin станет равным pend, мы узнаем, что массив кончился.

Перепишем эту программу так, чтобы начало и конец массива передавались параметрами

```
#inc1ude <iostream>

void ia_print( int *pbegin, int *pend )
{
    while ( pbegin != pend ) {
        cout << *pbegin << ' ';
        ++pbegin;
    }
}

int main()
{
    int ia[9] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
    ia_print( ia, ia + 9 );
```

в некую обобщенную функцию, которая умеет печатать массив любого размера:

```
}
```

Наша функция стала более универсальной, однако, она умеет работать только с массивами типа int. Есть способ снять и это ограничение: преобразовать данную

```
#inc1ude <iostream>

template <c1ass e1emType>
void print( elemType *pbegin, elemType *pend )
{
    while ( pbegin != pend ) {
        cout << *pbegin << ' ';
        ++pbegin;
    }
```

функцию в шаблон (шаблоны были вкратце представлены в разделе 2.5):

```
}
```

```
int main()
{
    int ia[9] =  { 0,  1, 1,  2, 3,  5, 8,  13, 21 };
    double da[4] = { 3.14,    6.28, 12.56, 25.12 };
    string sa[3] = { "piglet", "eeyore", "pooh" };

    print( ia, ia+9 );
    print( da, da+4 );
    print( sa, sa+3 );
```

Теперь мы можем вызывать нашу функцию print() для печати массивов любого типа:

```
}
```

Мы написали *обобщенную функцию*. Стандартная библиотека предоставляет набор обобщенных алгоритмов (мы уже упоминали об этом в разделе 3.4), реализованных подобным образом. Параметрами таких функций являются указатели на начало и конец массива, с которым они производят определенные действия. Вот, например, как выглядят

```
#include <a1gorithm>
int main()
{
    int ia[6] = { 107, 28, 3, 47, 104, 76 };
    string sa[3] = { "piglet", "eeyore", "pooh" };

    sort( ia, ia+6 );
    sort( sa, sa+3 );
```

вызовы обобщенного алгоритма сортировки:

```
};
```

(Мы подробно остановимся на обобщенных алгоритмах в главе 12; в Приложении будут приведены примеры их использования.)

В стандартной библиотеке C++ содержится набор классов, которые инкапсулируют использование контейнеров и указателей. (Об этом говорилось в разделе 2.8.) В следующем разделе мы займемся стандартным контейнерным типом vector, являющимся объектно-ориентированной реализацией массива.

3.10. Класс vector

Использование класса `vector` (см. раздел 2.8) является альтернативой применению встроенных массивов. Этот класс предоставляет гораздо больше возможностей, поэтому его использование предпочтительней. Однако встречаются ситуации, когда не обойтись без массивов встроенного типа. Одна из таких ситуаций – обработка передаваемых программе параметров командной строки, о чем мы будем говорить в разделе 7.8. Класс `vector`, как и класс `string`, является частью стандартной библиотеки C++.

Для использования вектора необходимо включить заголовочный файл:

```
#include <vector>
```

Существуют два абсолютно разных подхода к использованию вектора, назовем их *идиомой массива* и *идиомой STL*. В первом случае объект класса `vector` используется точно так же, как массив встроенного типа. Определяется вектор заданной размерности:

```
vector< int > ivec( 10 );
```

что аналогично определению массива встроенного типа:

```
int ia[ 10 ];
```

```
void simp1e_examp1e()
{
    const int e1em_size = 10;
    vector< int > ivec( e1em_size );
    int ia[ e1em_size ];

    for ( int ix = 0; ix < e1em_size; ++ix )
        ia[ ix ] = ivec[ ix ];

    // ...
```

Для доступа к отдельным элементам вектора применяется операция взятия индекса:

```
}
```

Мы можем узнать размерность вектора, используя функцию `size()`, и проверить, пуст ли

```
void print_vector( vector<int> ivec )
{
    if ( ivec.empty() )
        return;

    for ( int ix=0; ix< ivec.size(); ++ix )
        cout << ivec[ ix ] << ' ';
```

вектор, с помощью функции `empty()`. Например:

```
}
```

Элементы вектора инициализируются значениями по умолчанию. Для числовых типов и указателей таким значением является `0`. Если в качестве элементов выступают объекты

класса, то инициатор для них задается конструктором по умолчанию (см. раздел 2.3). Однако инициатор можно задать и явно, используя форму:

```
vector< int > ivec( 10, -1 );
```

Все десять элементов вектора будут равны -1.

Массив встроенного типа можно явно инициализировать списком:

```
int ia[ 6 ] = { -2, -1, 0, 1, 2, 1024 };
```

Для объекта класса vector аналогичное действие невозможно. Однако такой объект

```
// 6 элементов ia копируются в ivec
```

может быть инициализирован с помощью массива встроенного типа:

```
vector< int > ivec( ia, ia+6 );
```

Конструктору вектора ivec передаются два указателя – указатель на начало массива ia и на элемент, следующий за последним. В качестве списка начальных значений допустимо

```
// копируются 3 элемента: ia[2], ia[3], ia[4]
```

указать не весь массив, а некоторый его диапазон:

```
vector< int > ivec( &ia[ 2 ], &ia[ 5 ] );
```

Еще одним отличием вектора от массива встроенного типа является возможность инициализации одного объекта типа vector другим и использования операции

```
vector< string > svec;

void init_and_assign()
{
    // один вектор инициализируется другим
    vector< string > user_names( svec );
    // ...

    // один вектор копируется в другой
    svec = user_names;
}
```

присваивания для копирования объектов. Например:

Говоря об идиоме STL[6], мы подразумеваем совсем другой подход к использованию вектора. Вместо того чтобы сразу задать нужный размер, мы определяем пустой вектор:

```
vector< string > text;
```

---

6 STL расшифровывается как Standard Template Library. До появления стандартной библиотеки C++ классы vector, string и другие, а также обобщенные алгоритмы входили в отдельную библиотеку с названием STL.

Затем добавляем к нему элементы при помощи различных функций. Например, функция push_back()вставляет элемент в конец вектора. Вот фрагмент кода, считывающего

```
string word;
while ( cin >> word ) {
    text.push_back( word );
    // ...
```

последовательность строк из стандартного ввода и добавляющего их в вектор:

```
}
```

```
cout << "считаны слова: \n";
for ( int ix =0; ix < text.size(); ++ix )
    cout << text[ ix ] << ' ';
```

Хотя мы можем использовать операцию взятия индекса для перебора элементов вектора:

```
cout << endl;
```

```
cout << "считаны слова: \n";
for ( vector<string>::iterator it = text.begin();
      it != text.end(); ++it )
    cout << *it << ' ';
```

более типичным в рамках данной идиомы будет использование итераторов:

```
cout << endl;
```

Итератор – это класс стандартной библиотеки, фактически являющийся указателем на элемент массива.

Выражение

```
*it;
```

разыменовывает итератор и дает сам элемент вектора. Инструкция

```
++it;
```

сдвигает указатель на следующий элемент. Не нужно смешивать эти два подхода. Если следовать идиоме STL при определении пустого вектора:

```
vector<int> ivec;
```

будет ошибкой написать:

```
ivec[0] = 1024;
```

У нас еще нет ни одного элемента вектора ivec; количество элементов выясняется с помощью функции size().

Можно допустить и противоположную ошибку. Если мы определили вектор некоторого размера, например:

```
vector<int> ia( 10 );
```

то вставка элементов увеличивает его размер, добавляя новые элементы к существующим. Хотя это и кажется очевидным, тем не менее, начинающий программист

```
const int size = 7;
int ia[ size ] = { 0, 1, 1, 2, 3, 5, 8 };
vector< int > ivec( size );
for ( int ix = 0; ix < size; ++ix )
```

вполне мог бы написать:

```
    ivec.push_back( ia[ ix ] );
```

Имелась в виду инициализация вектора `ivec` значениями элементов `ia`, вместо чего получился вектор `ivec` размера `14`.

Следуя идиоме STL, можно не только добавлять, но и удалять элементы вектора. (Все это мы рассмотрим подробно и с примерами в главе 6.)

Упражнение 3.24

```
int ia[ 7 ] = { 0, 1, 1, 2, 3, 5, 8 };

(a) vector< vector< int > > ivec;
(b) vector< int > ivec = { 0, 1, 1, 2, 3, 5, 8 };
(c) vector< int > ivec( ia, ia+7 );
(d) vector< string > svec = ivec;
```

Имеются ли ошибки в следующих определениях?

```
(e) vector< string > svec( 10, string( "null" ));
```

Упражнение 3.25

```
bool is_equa1( const int*ia, int ia_size,
```

Реализуйте следующую функцию:

```
               const vector<int> &ivec );
```

Функция `is_equal()` сравнивает поэлементно два контейнера. В случае разного размера контейнеров “хвост” более длинного в расчет не принимается. Понятно, что, если все сравниваемые элементы равны, функция возвращает `true`, если отличается хотя бы один – `false`. Используйте итератор для перебора элементов. Напишите функцию `main()`, обращающуюся к `is_equal()`.

3.11. Класс complex

Класс комплексных чисел `complex` – еще один класс из стандартной библиотеки. Как обычно, для его использования нужно включить заголовочный файл:

```
#include <comp1ex>
```

Комплексное число состоит из двух частей – вещественной и мнимой. Мнимая часть представляет собой квадратный корень из отрицательного числа. Комплексное число принято записывать в виде

```
2 + 3i
```

где `2` – действительная часть, а `3i` – мнимая. Вот примеры определений объектов типа

```
// чисто мнимое число: 0 + 7-i
comp1ex< double > purei( 0, 7 );

// мнимая часть равна 0: 3 + Oi
comp1ex< float > rea1_num( 3 );

// и вещественная, и мнимая часть равны 0: 0 + 0-i
comp1ex< long double > zero;

// инициализация одного комплексного числа другим
```

complex:

```
comp1ex< double > purei2( purei );
```

Поскольку `complex`, как и `vector`, является шаблоном, мы можем конкретизировать его типами `float`, `double` и `long double`, как в приведенных примерах. Можно также

```
complex< double > conjugate[ 2 ] = {
    complex< double >( 2, 3 ),
    complex< double >( 2, -3 )
```

определить массив элементов типа `complex`:

```
};
```

```
complex< double > *ptr = &conjugate[0];
```

Вот как определяются указатель и ссылка на комплексное число:

```
complex< double > &ref = *ptr;
```

Комплексные числа можно складывать, вычитать, умножать, делить, сравнивать, получать значения вещественной и мнимой части. (Более подробно мы будем говорить о классе `complex` в разделе 4.6.)

3.12. Директива typedef

Директива `typedef` позволяет задать синоним для встроенного либо пользовательского типа данных. Например:

```
typedef double      wages;
typedef vector<int> vec_int;
typedef vec_int     test_scores;
typedef bool        in_attendance;

typedef int         *Pint;
```

Имена, определенные с помощью директивы typedef, можно использовать точно так же,

```
// double hourly, weekly;
wages hourly, weekly;

// vector<int> vecl( 10 );
vec_int vecl( 10 );

// vector<int> test0( c1ass_size );
const int c1ass_size = 34;
test_scores test0( c1ass_size );

// vector< bool > attendance;
vector< in_attendance > attendance( c1ass_size );

// int *table[ 10 ];
```

как спецификаторы типов:

```
Pint table [ 10 ];
```

Эта директива начинается с ключевого слова typedef, за которым идет спецификатор типа, и заканчивается идентификатором, который становится синонимом для указанного типа.

Для чего используются имена, определенные с помощью директивы typedef? Применяя мнемонические имена для типов данных, можно сделать программу более легкой для восприятия. Кроме того, принято употреблять такие имена для сложных составных типов, в противном случае воспринимаемых с трудом (см. пример в разделе 3.14), для объявления указателей на функции и функции-члены класса (см. раздел 13.6).

Ниже приводится пример вопроса, на который почти все дают неверный ответ. Ошибка вызвана непониманием директивы typedef как простой текстовой макроподстановки. Дано определение:

```
typedef char *cstring;
```

Каков тип переменной cstr в следующем объявлении:

```
extern const cstring cstr;
```

Ответ, который кажется очевидным:

```
const char *cstr
```

Однако это неверно. Спецификатор const относится к cstr, поэтому правильный ответ – константный указатель на char:

```
char *const cstr;
```

3.13. Спецификатор volatile

Объект объявляется как volatile (неустойчивый, асинхронно изменяемый), если его значение может быть изменено незаметно для компилятора, например переменная, обновляемая значением системных часов. Этот спецификатор сообщает компилятору, что не нужно производить оптимизацию кода для работы с данным объектом.

```
volatile int disp1ay_register;
volatile Task *curr_task;
volatile int ixa[ max_size ];
```

Спецификатор volatile используется подобно спецификатору const:

```
volatile Screen bitmap_buf;
```

display_register – неустойчивый объект типа int. curr_task – указатель на неустойчивый объект класса Task. ixa – неустойчивый массив целых, причем каждый элемент такого массива считается неустойчивым. bitmap_buf – неустойчивый объект класса Screen, каждый его член данных также считается неустойчивым.

Единственная цель использования спецификатора volatile – сообщить компилятору, что тот не может определить, кто и как может изменить значение данного объекта. Поэтому компилятор не должен выполнять оптимизацию кода, использующего данный объект.

3.14. Класс pair

Класс pair (пара) стандартной библиотеки C++ позволяет нам определить одним объектом пару значений, если между ними есть какая-либо семантическая связь. Эти значения могут быть одинакового или разного типа. Для использования данного класса необходимо включить заголовочный файл:

```
#inc1ude <uti1ity>
```

Например, инструкция

```
pair< string, string > author( "James", "Joyce" );
```

создает объект author типа pair, состоящий из двух строковых значений.

```
string firstBook;

if ( Joyce.first == "James" &&
     Joyce.second == "Joyce" )
```

Отдельные части пары могут быть получены с помощью членов first и second:

```
  firstBook = "Stephen Hero";
```

Если нужно определить несколько однотипных объектов этого класса, удобно использовать директиву typedef:

```
typedef pair< string, string > Authors;

Authors proust( "marcel", "proust" );
Authors joyce( "James", "Joyce" );

Authors musil( "robert", "musi1" );
```

Вот другой пример употребления пары. Первое значение содержит имя некоторого

```
class EntrySlot;
extern EntrySlot* 1ook_up( string );

typedef pair< string, EntrySlot* > SymbolEntry;

SymbolEntry current_entry( "author", 1ook_up( "author" ));
// ...

if ( EntrySlot *it = 1ook_up( "editor" ))
{
    current_entry.first = "editor";
    current_entry.second = it;
}
```

объекта, второе – указатель на соответствующий этому объекту элемент таблицы.

(Мы вернемся к рассмотрению класса `pair` в разговоре о контейнерных типах в главе 6 и об обобщенных алгоритмах в главе 12.)

3.15. Типы классов

Механизм классов позволяет создавать новые типы данных; с его помощью введены типы `string`, `vector`, `complex` и `pair`, рассмотренные выше. В главе 2 мы рассказывали о концепциях и механизмах, поддерживающих объектный и объектно-ориентированный подход, на примере реализации класса `Array`. Здесь мы, основываясь на объектном подходе, создадим простой класс `String`, реализация которого поможет понять, в частности, *перегрузку операций* – мы говорили о ней в разделе 2.3. (Классы подробно рассматриваются в главах 13, 14 и 15. Мы дали краткое описание класса для того, чтобы приводить более интересные примеры. Читатель, только начинающий изучение C++, может пропустить этот раздел и подождать более систематического описания классов в следующих главах.)

Наш класс `String` должен поддерживать инициализацию объектом класса `String`, строковым литералом и встроенным строковым типом, равно как и операцию присваивания ему значений этих типов. Мы используем для этого конструкторы класса и перегруженную операцию присваивания. Доступ к отдельным символам `String` будет реализован как перегруженная операция взятия индекса. Кроме того, нам понадобятся: функция `size()` для получения информации о длине строки; операция сравнения объектов типа `String` и объекта `String` со строкой встроенного типа; а также операции ввода/вывода нашего объекта. В заключение мы реализуем возможность доступа к внутреннему представлению нашей строки в виде строки встроенного типа.

Определение класса начинается ключевым словом `class`, за которым следует идентификатор – имя класса, или типа. В общем случае класс состоит из секций, предваряемых словами `public` (открытая) и `private` (закрытая). Открытая секция, как

правило, содержит набор операций, поддерживаемых классом и называемых методами или функциями-членами класса. Эти функции-члены определяют открытый интерфейс класса, другими словами, набор действий, которые можно совершать с объектами данного класса. В закрытую секцию обычно включают данные-члены, обеспечивающие внутреннюю реализацию. В нашем случае к внутренним членам относятся _string – указатель на char, а также _size типа int. _size будет хранить информацию о длине строки, а _string – динамически выделенный массив символов. Вот как выглядит

```
#inc1ude <iostream>

class String;
istream& operator>>( istream&, String& );
ostream& operator<<( ostream&, const String& );

class String {
public:
  // набор конструкторов
  // для автоматической инициализации
  // String strl;             // String()
  // String str2( "literal" ); // String( const char* );
  // String str3( str2 );      // String( const String& );

  String();
  String( const char* );
  String( const String& );

  // деструктор
  ~String();

  // операторы присваивания
  // strl = str2
  // str3 = "a string literal"

  String& operator=( const String& );
  String& operator=( const char* );

  // операторы проверки на равенство
  // strl == str2;
  // str3 == "a string literal";

  bool operator==( const String& );
  bool operator==( const char* );

  // перегрузка оператора доступа по индексу
  // strl[ 0 ] = str2[ 0 ];

  char& operator[]( int );

  // доступ к членам класса
  int   size() { return _size; }
  char* c_str() { return _string; }

private:
  int  _size;
  char *_string;
```

определение класса:

```
}
```

Класс `String` имеет три конструктора. Как было сказано в разделе 2.3, механизм перегрузки позволяет определять несколько реализаций функций с одним именем, если все они различаются количеством и/или типами своих параметров. Первый конструктор

```
String();
```

является конструктором по умолчанию, потому что не требует явного указания начального значения. Когда мы пишем:

```
String str1;
```

для `str1` вызывается такой конструктор.

Два оставшихся конструктора имеют по одному параметру. Так, для

```
String str2("строка символов");
```

вызывается конструктор

```
String(const char*);
```

а для

```
String str3(str2);
```

конструктор

```
String(const String&);
```

Тип вызываемого конструктора определяется типом фактического аргумента. Последний из конструкторов, `String(const String&)`, называется *копирующим*, так как он инициализирует объект копией другого объекта.

Если же написать:

```
String str4(1024);
```

то это вызовет ошибку компиляции, потому что нет ни одного конструктора с параметром типа `int`.

Объявление перегруженного оператора имеет следующий формат:

```
return_type operator op (parameter_list);
```

где `operator` – ключевое слово, а `op` – один из предопределенных операторов: `+`, `=`, `==`, `[]` и так далее. (Точное определение синтаксиса см. в главе 15.) Вот объявление перегруженного оператора взятия индекса:

```
char& operator[] (int);
```

Этот оператор имеет единственный параметр типа `int` и возвращает ссылку на `char`. Перегруженный оператор сам может быть перегружен, если списки параметров

отдельных конкретизаций различаются. Для нашего класса String мы создадим по два различных оператора присваивания и проверки на равенство.

Для вызова функции-члена применяются операторы доступа к членам – точка (.) или

```
String object("Danny");
String *ptr = new String ("Anna");
```

стрелка (->). Пусть мы имеем объявления объектов типа String:

```
String array[2];
```

```
vector<int> sizes( 3 );

// доступ к члену для objects (.);
// objects имеет размер 5
sizes[ 0 ] = object.size();

// доступ к члену для pointers (->)
// ptr имеет размер 4
sizes[ 1 ] = ptr->size();

// доступ к члену (.)
// array[0] имеет размер 0
```

Вот как выглядит вызов функции size() для этих объектов:

```
sizes[ 2 ] = array[0].size();
```

Она возвращает соответственно 5, 4 и 0.

```
String name1( "Yadie" );
String name2( "Yodie" );

// bool operator==(const String&)
if ( name1 == name2 )
    return;
else
// String& operator=( const String& )
```

Перегруженные операторы применяются к объекту так же, как обычные:

```
    name1 = name2;
```

Объявление функции-члена должно находиться внутри определения класса, а определение функции может стоять как внутри определения класса, так и вне его. (Обе функции size() и c_str() определяются внутри класса.) Если функция определяется вне класса, то мы должны указать, кроме всего прочего, к какому классу она принадлежит. В этом случае определение функции помещается в исходный файл, допустим, String.C, а определение самого класса – в заголовочный файл (String.h в нашем примере), который должен включаться в исходный:

```
// содержимое исходного файла: String.C

// включение определения класса String
#inc1ude "String.h"

// включение определения функции strcmp()
#inc1ude <cstring>
bool              // тип возвращаемого значения
String::    // класс, которому принадлежит функция
operator==  // имя функции: оператор равенства
(const String &rhs)  // список параметров
{
    if ( _size != rhs._size )
        return false;
    return strcmp( _strinq, rhs._string ) ?
        false : true;
}
```

Напомним, что `strcmp()` – функция стандартной библиотеки C. Она сравнивает две строки встроенного типа, возвращая 0 в случае равенства строк и ненулевое значение в случае неравенства. *Условный оператор* (`?:`) проверяет значение, стоящее перед знаком вопроса. Если оно истинно, возвращается значение выражения, стоящего слева от двоеточия, в противном случае – стоящего справа. В нашем примере значение выражения равно `false`, если `strcmp()` вернула ненулевое значение, и `true` – если нулевое. (Условный оператор рассматривается в разделе 4.7.)

Операция сравнения довольно часто используется, реализующая ее функция получилась небольшой, поэтому полезно объявить эту функцию встроенной (`inline`). Компилятор подставляет текст функции вместо ее вызова, поэтому время на такой вызов не затрачивается. (Встроенные функции рассматриваются в разделе 7.6.) Функция-член, определенная внутри класса, является встроенной по умолчанию. Если же она определена

```
inline bool
String::operator==(const String &rhs)
{
    // то же самое
}
```

вне класса, чтобы объявить ее встроенной, нужно употребить ключевое слово `inline`:

Определение встроенной функции должно находиться в заголовочном файле, содержащем определение класса. Переопределив оператор `==` как встроенный, мы должны переместить сам текст функции из файла `String.C` в файл `String.h`.

Ниже приводится реализация операции сравнения объекта `String` со строкой

```
inline bool
String::operator==(const char *s)
{
    return strcmp( _string, s ) ? false : true;
}
```

встроенного типа:

Имя конструктора совпадает с именем класса. Считается, что он не возвращает значение, поэтому не нужно задавать возвращаемое значение ни в его определении, ни в его теле. Конструкторов может быть несколько. Как и любая другая функция, они могут быть

```
#include <cstring>

// default constructor
inline String::String()
{
    _size = 0;
    _string = 0;
}

inline String::String( const char *str )
{
    if ( ! str ) {
        _size = 0; _string = 0;
    }
    else {
        _size = str1en( str );
        _string = new char[ _size + 1 ];
        strcpy( _string, str );
    }

// copy constructor
inline String::String( const  String &rhs  )
{
    size = rhs._size;
    if ( ! rhs._string  )
        _string = 0;
    else {
        _string = new char[ _size + 1 ];
        strcpy( _string, rhs._string );
   }
```

объявлены встроенными.

```
}
```

Поскольку мы динамически выделяли память с помощью оператора new, необходимо освободить ее вызовом delete, когда объект String нам больше не нужен. Для этой цели служит еще одна специальная функция-член – деструктор, автоматически вызываемый для объекта в тот момент, когда этот объект перестает существовать. (См. главу 7 о времени жизни объекта.) Имя деструктора образовано из символа тильды (~) и имени класса. Вот определение деструктора класса String. Именно в нем мы вызываем операцию delete, чтобы освободить память, выделенную в конструкторе:

```
inline String: :~String() { delete [] _string; }
```

В обоих перегруженных операторах присваивания используется специальное ключевое слово this.

```
String namel( "orville" ), name2( "wilbur" );
```

Когда мы пишем:

```
namel = "Orville Wright";
```

this является указателем, адресующим объект name1 внутри тела функции операции присваивания.

```
ptr->size();
```

this всегда указывает на объект класса, через который происходит вызов функции. Если

```
obj[ 1024 ];
```

то внутри size() значением this будет адрес, хранящийся в ptr. Внутри операции взятия индекса this содержит адрес obj. Разыменовывая this (использованием *this),

```
inline String&
String::operator=( const char *s )
{
    if ( ! s ) {
        _size = 0;
        delete [] _string;
        _string = 0;
    }
    else {
        _size = str1en( s );
        delete [] _string;
        _string = new char[ _size + 1 ];
        strcpy( _string, s );
    }
```

мы получаем сам объект. (Указатель this детально описан в разделе 13.4.)

```
    return *this;
}
```

При реализации операции присваивания довольно часто допускают одну ошибку: забывают проверить, не является ли копируемый объект тем же самым, в который происходит копирование. Мы выполним эту проверку, используя все тот же указатель

```
inline String&
String::operator=( const String &rhs )
{
    // в выражении
    // namel = *pointer_to_string
    // this представляет собой name1,
    // rhs - *pointer_to_string.
```

this:

```
    if ( this != &rhs ) {
```

Вот полный текст операции присваивания объекту String объекта того же типа:

```
inline String&
String::operator=( const String &rhs )
{
    if ( this != &rhs ) {
        delete [] _string;
        _size = rhs._size;

        if ( ! rhs._string )
            _string = 0;
        else {
            _string = new char[ _size + 1 ];
            strcpy( _string, rhs._string );
        }
    }
    return *this;
}
```

Операция взятия индекса практически совпадает с ее реализацией для массива `Array`,

```
#include <cassert>

inline char&
String::operator[] ( int elem )
{
    assert( elem >= 0 && elem < _size );
    return _string[ elem ];
}
```

который мы создали в разделе 2.3:

Операторы ввода и вывода реализуются как отдельные функции, а не члены класса. (О причинах этого мы поговорим в разделе 15.2. В разделах 20.4 и 20.5 рассказывается о перегрузке операторов ввода и вывода библиотеки `iostream`.) Наш оператор ввода может прочесть не более 4095 символов. `setw()` – предопределенный манипулятор, он читает из входного потока заданное число символов минус 1, гарантируя тем самым, что мы не переполним наш внутренний буфер `inBuf`. (В главе 20 манипулятор `setw()` рассматривается детально.) Для использования манипуляторов нужно включить

```
#include <iomanip>

inline istream&
operator>>( istream &io, String &s )
{
    // искусственное ограничение: 4096 символов
    const int 1imit_string_size = 4096;
    char inBuf[ limit_string_size ];
    // setw() входит в библиотеку iostream
    // он ограничивает размер читаемого блока до 1imit_string_size-l
    io >> setw( 1imit_string_size ) >> inBuf;
    s = mBuf; // String::operator=( const char* );
    return io;
}
```

соответствующий заголовочный файл:

Оператору вывода необходим доступ к внутреннему представлению строки String. Так как operator<< не является функцией-членом, он не имеет доступа к закрытому члену данных _string. Ситуацию можно разрешить двумя способами: объявить operator<< *дружественным* классу String, используя ключевое слово friend (дружественные отношения рассматриваются в разделе 15.2), или реализовать встраиваемую (inline) функцию для доступа к этому члену. В нашем случае уже есть такая функция: c_str() обеспечивает доступ к внутреннему представлению строки. Воспользуемся ею при

```
inline ostream&
operator<<( ostream& os, const String &s )
{
    return os << s.c_str();
}
```

реализации операции вывода:

Ниже приводится пример программы, использующей класс String. Эта программа берет слова из входного потока и подсчитывает их общее число, а также количество слов "the" и "it" и регистрирует встретившиеся гласные.

```
#include <iostream>
#inc1ude "String.h"
int main() {
    int aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0,
        theCnt = 0, itCnt = 0, wdCnt = 0, notVowel = 0;

    // Слова "The" и "It"
    // будем проверять с помощью operator==( const char* )
    String but, the( "the" ), it( "it" );

    // operator>>( ostream&, String& )
    while ( cin >> buf ) {
        ++wdCnt;

        // operator<<( ostream&, const String& )
        cout << buf << ' ';

        if ( wdCnt % 12 == 0 )
            cout << endl;

        // String::operator==( const String& ) and
        // String::operator==( const char* );
        if ( buf == the | | buf == "The"   )
            ++theCnt;
        else
        if ( buf == it || buf == "It" )
            ++itCnt;

        // invokes String::s-ize()
        for ( int ix =0; ix < buf.sizeO; ++ix )
        {
            // invokes String:: operator [] (int)
            switch( buf[ ix ] )
            {
                case 'a': case 'A': ++aCnt; break;
                case 'e': case 'E': ++eCnt; break;
                case 'i': case 'I': ++iCnt; break;
                case 'o': case '0': ++oCnt; break;
                case 'u': case 'U': ++uCnt; break;
                default: ++notVowe1; break;
            }
        }
    }

    // operator<<( ostream&, const   String& )
    cout << "\n\n"
         << "Слов: " << wdCnt  << "\n\n"
         << "the/The: " << theCnt << '\n'
         << "it/It: " << itCnt  << "\n\n"
         << "согласных: " < <notVowel << "\n\n"
         << "a: " << aCnt << '\n'
         << "e: " << eCnt << '\n'
         << "i: " << ICnt << '\n'
         << "o: " << oCnt << '\n'
         << "u: " << uCnt << endl;

}
```

Протестируем программу: предложим ей абзац из детского рассказа, написанного одним из авторов этой книги (мы еще встретимся с этим рассказом в главе 6). Вот результат работы программы:

```
Alice Emma has long flowing red hair. Her Daddy says when the
wind blows through her hair, it looks almost alive, 1ike a fiery
bird in flight. A beautiful fiery bird, he tells her, magical but
untamed. "Daddy, shush, there is no such thing," she tells him, at
the same time wanting him to tell her more. Shyly, she asks,
"I mean, Daddy, is there?"

Слов: 65
the/The: 2
it/It: 1
согласных: 190
a: 22
e: 30
i: 24
o: 10
u: 7
```

Упражнение 3.26

В наших реализациях конструкторов и операций присваивания содержится много повторов. Попробуйте вынести повторяющийся код в отдельную закрытую функцию-член, как это было сделано в разделе 2.3. Убедитесь, что новый вариант работоспособен.

Упражнение 3.27

Модифицируйте тестовую программу так, чтобы она подсчитывала и согласные b, d, f, s, t.

Упражнение 3.28

Напишите функцию-член, подсчитывающую количество вхождений символа в строку

```
class String {
public:
    // ...
    int count( char ch  ) const;
    // ...
};
```

`String`, используя следующее объявление:

Упражнение 3.29

Реализуйте оператор конкатенации строк (`+`) так, чтобы он конкатенировал две строки и

```
class String {
public:
    // ...
    String operator+( const String &rhs ) const;
    // ...
};
```

возвращал результат в новом объекте `String`. Вот объявление функции:

4

4. Выражения

В главе 3 мы рассмотрели типы данных – как встроенные, так и предоставленные стандартной библиотекой. Здесь мы разберем предопределенные операции, такие, как сложение, вычитание, сравнение и т.п., рассмотрим их приоритеты. Скажем, результатом выражения 3+4*5 является 23, а не 35 потому, что операция умножения (*) имеет более высокий приоритет, чем операция сложения (+). Кроме того, мы обсудим вопросы преобразований типов данных – и явных, и неявных. Например, в выражении 3+0.7 целое значение 3 станет вещественным перед выполнением операции сложения.

4.1. Что такое выражение?

Выражение состоит из одного или более операндов, в простейшем случае – из одного литерала или объекта. Результатом такого выражения является r-значение его операнда.

```
void mumble() {
    3.14159;
    "melancholia";
    upperBound;
}
```

Например:

Результатом вычисления выражения `3.14159` станет `3.14159` типа `double`, выражения `"melancholia"` – адрес первого элемента строки типа `const char*`. Значение выражения `upperBound` – это значение объекта `upperBound`, а его типом будет тип самого объекта.

Более общим случаем выражения является один или более операндов и некоторая

```
salary + raise
ivec[ size/2 ] * delta
```

операция, применяемая к ним:

```
first_name + " " + 1ast_name
```

Операции обозначаются соответствующими знаками. В первом примере сложение применяется к `salary` и `raise`. Во втором выражении `size` делится на `2`. Частное используется как индекс для массива `ivec`. Получившийся в результате операции взятия индекса элемент массива умножается на `delta`. В третьем примере два строковых объекта конкатенируются между собой и со строковым литералом, создавая новый строковый объект.

Операции, применяемые к одному операнду, называются *унарными* (например, взятие адреса (&) и разыменование (*)), а применяемые к двум операндам – *бинарными*. Один и

тот же символ может обозначать разные операции в зависимости от того, унарна она или бинарна. Так, в выражении

```
*ptr
```

* представляет собой унарную операцию разыменования. Значением этого выражения является значение объекта, адрес которого содержится в ptr. Если же написать:

```
var1 * var2
```

то звездочка будет обозначать бинарную операцию умножения.

Результатом вычисления выражения всегда, если не оговорено противное, является r-значение. Тип результата арифметического выражения определяется типами операндов. Если операнды имеют разные типы, производится преобразование типов в соответствии с предопределенным набором правил. (Мы детально рассмотрим эти правила в разделе 4.14.)

Выражение может являться составным, то есть объединять в себе несколько подвыражений. Вот, например, выражение, проверяющее на неравенство нулю указатель и объект, на который он указывает (если он на что-то указывает)7:

```
ptr != 0 && *ptr != 0
```

Выражение состоит из трех подвыражений: проверку указателя ptr, разыменования ptr

```
int ival = 1024;
```

и проверку результата разыменования. Если ptr определен как

```
int *ptr = &ival;
```

то результатом разыменования будет 1024 и оба сравнения дадут истину. Результатом всего выражения также будет истина (оператор && обозначает логическое И).

Если посмотреть на этот пример внимательно, можно заметить, что порядок выполнения операций очень важен. Скажем, если бы операция разыменования ptr производилась до его сравнения с 0, в случае нулевого значения ptr это скорее всего вызвало бы крах программы. В случае операции И порядок действий строго определен: сначала оценивается левый операнд, и если его значение равно false, правый операнд не вычисляется вовсе. Порядок выполнения операций определяется их приоритетами, не всегда очевидными, что вызывает у начинающих программистов на C и C++ множество ошибок. Приоритеты будут приведены в разделе 4.13, а пока мы расскажем обо всех операциях, определенных в C++, начиная с наиболее привычных.

4.2. Арифметические операции

Таблица 4.1. Арифметические операции

---

7 Проверку на неравенство 0 можно опустить. Полностью эквивалентна приведенной и более употребима следующая запись: ptr && *ptr.

Символ операции	Значение	Использование
*	Умножение	expr * expr
/	Деление	expr / expr
%	Остаток от деления	expr % expr
+	Сложение	expr + expr
-	Вычитание	expr – expr

Деление целых чисел дает в результате целое число. Дробная часть результата, если она

```
int ivall = 21 / 6;
```

есть, отбрасывается:

```
int iva12 = 21 / 7;
```

И `ival1`, и `ival2` в итоге получат значение 3.

Операция *остаток* (%), называемая также делением по модулю, возвращает остаток от деления первого операнда на второй, но применяется только к операндам целого типа (char, short, int, long). Результат положителен, если оба операнда положительны. Если же один или оба операнда отрицательны, результат зависит от реализации, то есть машинно-зависим. Вот примеры правильного и неправильного использования деления по

```
3.14 % 3; // ошибка: операнд типа double
21 % 6;   // правильно:  3
21 % 7;   // правильно:  0
21 % -5;  // машинно-зависимо: -1 или 1

int iva1 = 1024;
double dval = 3.14159;

iva1 % 12;   // правильно:
```

модулю:

```
iva1 % dval; // ошибка: операнд типа double
```

Иногда результат вычисления арифметического выражения может быть неправильным либо не определенным. В этих случаях говорят об арифметических исключениях (хотя они не вызывают возбуждения исключения в программе). Арифметические исключения могут иметь чисто математическую природу (скажем, деление на 0) или происходить от представления чисел в компьютере – как *переполнение* (когда значение превышает величину, которая может быть выражена объектом данного типа). Например, тип char содержит 8 бит и способен хранить значения от 0 до 255 либо от -128 до 127 в зависимости от того, знаковый он или беззнаковый. В следующем примере попытка присвоить объекту типа char значение 256 вызывает переполнение:

```
#include <iostream>

int main() {
    char byte_value = 32;
    int ival = 8;

    // переполнение памяти, отведенной под byte_value
    byte_value = ival * byte_value;

    cout << "byte_value: " <<static_cast<int>(byte_value) << endl;
}
```

Для представления числа 256 необходимы 9 бит. Переменная byte_value получает некоторое неопределенное (машинно-зависимое) значение. Допустим, на нашей рабочей станции SGI мы получили 0. Первая попытка напечатать это значение с помощью:

```
cout << "byte_va1ue: " << byte_va1ue << endl;
```

привела к результату:

```
byte_value:
```

После некоторого замешательства мы поняли, что значение 0 – это нулевой символ ASCII, который не имеет представления при печати. Чтобы напечатать не представление символа, а его значение, нам пришлось использовать весьма странно выглядящее выражение:

```
static_cast<int>(byte_value)
```

которое называется явным приведением типа. Оно преобразует тип объекта или выражения в другой тип, явно заданный программистом. В нашем случае мы изменили byte_value на int. Теперь программа выдает более осмысленный результат:

```
byte_value: 0
```

На самом деле нужно было изменить не значение, соответствующее byte_value, а поведение операции вывода, которая действует по-разному для разных типов. Объекты типа char представляются ASCII-символами (а не кодами), в то время как для объектов типа int мы увидим содержащиеся в них значения. (Преобразования типов рассмотрены в разделе 4.14.)

Это небольшое отступление от темы – обсуждение проблем преобразования типов – вызвано обнаруженной нами погрешностью в работе нашей программы и в каком-то смысле напоминает реальный процесс программирования, когда аномальное поведение программы заставляет на время забыть о том, ради достижения какой, собственно, цели она пишется, и сосредоточиться на несущественных, казалось бы, деталях. Такая мелочь, как недостаточно продуманный выбор типа данных, приводящий к переполнению, может стать причиной трудно обнаруживаемой ошибки: из соображений эффективности проверка на переполнение не производится во время выполнения программы.

Стандартная библиотека C++ имеет заголовочный файл `limits`, содержащий различную информацию о встроенных типах данных, в том числе и диапазоны значений для каждого типа. Заголовочные файлы `climits` и `cfloat` также содержат эту информацию. (Об использовании этих заголовочных файлов для того, чтобы избежать переполнения и потери значимости, см. главы 4 и 6 [PLAUGER92]).

Примечание [O.A.1]: Как должны быть оформлены ссылки на книги, указанные в библиографии? Пришлите ваши пожелания.

Арифметика вещественных чисел создает еще одну проблему, связанную с *округлением*. Вещественное число представляется фиксированным количеством разрядов (разным для разных типов – `float`, `double` и `long double`), и точность значения зависит от используемого типа данных. Но даже самый точный тип `long double` не может устранить ошибку округления. Вещественная величина в любом случае представляется с некоторой ограниченной точностью. (См. [SHAMPINE97] о проблемах округления вещественных чисел.)

Упражнение 4.1

```
double dvall = 10.0, dva12 = 3.0;
int ivall = 10, iva12 = 3;

dvall / dva12;
```

В чем разница между приведенными выражениями с операцией деления?

```
ivall / iva12;
```

Упражнение 4.2

Напишите выражение, определяющее, четным или нечетным является данное целое число.

Упражнение 4.3

Найдите заголовочные файлы `limits`, `climits` и `cfloat` и посмотрите, что они содержат.

4.3. Операции сравнения и логические операции

Таблица 4.2. Операции сравнения и логические операции

Символ операции	Значение	Использование
!	Логическое НЕ	`!expr`
<	Меньше	`expr1 < expr2`
<=	Меньше или равно	`expr1 <= expr2`
>	Больше	`expr1 > expr2`
>=	Больше или равно	`expr1 >= expr2`
==	Равно	`expr1 == expr2`
!=	Не равно	`expr1 != expr2`
&&	Логическое И	`expr1 && expr2`
||	Логическое ИЛИ	`expr1 || expr2`
Примечание. Все операции в результате дают значение типа `bool`		

Операции сравнения и логические операции в результате дают значение типа bool, то есть true или false. Если же такое выражение встречается в контексте, требующем целого значения, true преобразуется в 1, а false – в 0. Вот фрагмент кода, подсчитывающего количество элементов вектора, меньших некоторого заданного

```
vector<int>::iterator iter = ivec.beg-in() ;
while ( iter != ivec.end() ) {
    // эквивалентно: e1em_cnt = e1em_cnt + (*iter < some_va1ue)
    // значение true/false выражения *iter < some_va1ue
    // превращается в 1 или 0
    e1em_cnt += *iter < some_va1ue;
    ++iter;
```

значения:

```
}
```

Мы просто прибавляем результат операции “меньше” к счетчику. (Пара += обозначает составной оператор присваивания, который складывает операнд, стоящий слева, и операнд, стоящий справа. То же самое можно записать более компактно: elem_count = elem_count + n. Мы рассмотрим такие операторы в разделе 4.4.)

Логическое И (&&) возвращает истину только тогда, когда истинны оба операнда. Логическое ИЛИ (||) дает истину, если истинен хотя бы один из операндов. Гарантируется, что операнды вычисляются слева направо и вычисление заканчивается, как только результирующее значение становится известно. Что это значит? Пусть даны

```
expr1 && expr2
```

два выражения:

```
expr1 || expr2
```

Если в первом из них expr1 равно false, значение всего выражения тоже будет равным false вне зависимости от значения expr2, которое даже не будет вычисляться. Во втором выражении expr2 не оценивается, если expr1 равно true, поскольку значение всего выражения равно true вне зависимости от expr2.

Подобный способ вычисления дает возможность удобной проверки нескольких

```
while ( ptr != 0 &&
        ptr->va1ue < upperBound &&
        ptr->va1ue >= 0 &&
        notFound( ia[ ptr->va1ue ] ))
```

выражений в одном операторе AND:

```
{ ... }
```

Указатель с нулевым значением не указывает ни на какой объект, поэтому применение к нулевому указателю операции доступа к члену вызвало бы ошибку (ptr->value). Однако, если ptr равен 0, проверка на первом шаге прекращает дальнейшее вычисление подвыражений. Аналогично на втором и третьем шагах проверяется попадание величины

ptr->value в нужный диапазон, и операция взятия индекса не применяется к массиву ia, если этот индекс неправилен.

Операция логического НЕ дает true, если ее единственный оператор равен false, и

```
bool found = false;
// пока элемент не найден
// и ptr указывает на объект (не 0)
while ( ! found && ptr ) {
    found = lookup( *ptr );
    ++ptr;
}
```

наоборот. Например:

Подвыражение

```
! found
```

дает true, если переменная found равна false. Это более компактная запись для

```
found == false
```

Аналогично

```
if ( found )
```

эквивалентно более длинной записи

```
if ( found == true )
```

Использование операций сравнения достаточно очевидно. Нужно только иметь в виду, что, в отличие от И и ИЛИ, порядок вычисления операндов таких выражений не

```
// Внимание! Порядок вычислений не определен!
if ( ia[ index++ ] < ia[ index ] )
```

определен. Вот пример, где возможна подобная ошибка:

```
    // поменять местами элементы
```

Программист предполагал, что левый операнд оценивается первым и сравниваться будут элементы ia[0] и ia[1]. Однако компилятор не гарантирует вычислений слева направо, и в таком случае элемент ia[0] может быть сравнен сам с собой. Гораздо лучше

```
if ( ia[ index ] < ia[ index+1 ] )
    // поменять местами элементы
```

написать более понятный и машинно-независимый код:

```
    ++index;
```

Еще один пример возможной ошибки. Мы хотели убедиться, что все три величины ival,

```
// Внимание! это не сравнение 3 переменных друг с другом
if ( ival != jva1 != kva1 )
```

jval и kval различаются. Где мы промахнулись?

```
    // do something ...
```

Значения 0, 1 и 0 дают в результате вычисления такого выражения true. Почему? Сначала проверяется ival != jval, а потом итог этой проверки (true/false –

```
if ( ival != jva1 && ival != kva1 && jva1 != kva1 )
```

преобразованной к 1/0) сравнивается с kval. Мы должны были явно написать:

```
    // сделать что-то ...
```

Упражнение 4.4

Найдите неправильные или непереносимые выражения, поясните. Как их можно

```
(a) ptr->iva1 != 0
(c) ptr != 0 && *ptr++
(e) vec[ iva1++ ] <= vec[ ival ];
```

изменить? (Заметим, что типы объектов не играют роли в данных примерах.)

```
(b) ival != jva1 < kva1 (d) iva1++ && ival
```

Упражнение 4.5

Язык C++ не диктует порядок вычисления операций сравнения для того, чтобы позволить компилятору делать это оптимальным образом. Как вы думаете, стоило бы в данном случае пожертвовать эффективностью, чтобы избежать ошибок, связанных с предположением о вычислении выражения слева направо?

4.4. Операции присваивания

```
int ival = 1024;
```

Инициализация задает начальное значение переменной. Например:

```
int *pi = 0;
```

В результате операции присваивания объект получает новое значение, при этом старое

```
ival = 2048;
```

пропадает:

```
pi = &iva1;
```

Иногда путают инициализацию и присваивание, так как они обозначаются одним и тем же знаком =. Объект инициализируется только один раз – при его определении. В то же время операция может быть применена к нему многократно.

Что происходит, если тип объекта не совпадает с типом значения, которое ему хотят присвоить? Допустим,

```
ival = 3.14159; // правильно?
```

В таком случае компилятор пытается трансформировать тип объекта, стоящего справа, в тип объекта, стоящего слева. Если такое преобразование возможно, компилятор неявно изменяет тип, причем при потере точности обычно выдается предупреждение. В нашем случае вещественное значение 3.14159 преобразуется в целое значение 3, и это значение присваивается переменной ival.

Если неявное приведение типов невозможно, компилятор сигнализирует об ошибке:

```
pi = ival; // ошибка
```

Неявная трансформация типа int в тип указатель на int невозможна. (Набор допустимых неявных преобразований типов мы обсудим в разделе 4.14.)

Левый операнд операции присваивания должен быть l-значением. Очевидный пример неправильного присваивания:

```
1024 = ival; // ошибка
```

```
int value = 1024;
```

Возможно, имелось в виду следующее:

```
value = ival; // правильно
```

Однако недостаточно потребовать, чтобы операнд слева от знака присваивания был l-

```
const int array_size = 8;
int ia[ array_size ] = { 0, 1, 2, 2, 3, 5, 8, 13 };
```

значением. Так, после определений

```
int *pia = ia;
```

выражение

```
array_size = 512; // ошибка
```

ошибочно, хотя array_size и является l-значением: объявление array_size константой не дает возможности изменить его значение. Аналогично

```
ia = pia; // ошибка
```

ia – тоже l-значение, но оно не может быть значением массива.

Неверна и инструкция

```
pia + 2=1; // ошибка
```

Хотя pia+2 дает адрес ia[2], присвоить ему значение нельзя. Если мы хотим изменить элемент ia[2], то нужно воспользоваться операцией разыменования. Корректной будет следующая запись:

```
*(pia + 2) = 1; // правильно
```

Операция присваивания имеет результат – значение, которое было присвоено самому левому операнду. Например, результатом такой операции

```
ival = 0;
```

является 0, а результат

```
ival = 3.14159;
```

равен 3. Тип результата – int в обоих случаях. Это свойство операции присваивания

```
extern char next_char();
int main()
{
    char ch = next_char();
    while ( ch != '\n' ) {
        // сделать что-то ...
        ch = next_char();
    }
    // ...
```

можно использовать в подвыражениях. Например, следующий цикл

```
}
```

```
extern char next_char();
int main()
{
    char ch;
    while (( ch = next_char() ) != '\n' ) {
        // сделать что-то ...
    }
    // ...
```

может быть переписан так:

```
}
```

Заметим, что вокруг выражения присваивания необходимы скобки, поскольку приоритет этой операции ниже, чем операции сравнения. Без скобок первым выполняется сравнение:

```
next_char() != '\n'
```

и его результат, true или false, присваивается переменной ch. (Приоритеты операций будут рассмотрены в разделе 4.13.)

Аналогично несколько операций присваивания могут быть объединены, если это

```
int main ()
{
    int ival, jval;
    ival = jval = 0; // правильно: присваивание 0 обеим переменным
```

позволяют типы операндов. Например:

```
    // ...
}
```

Обеим переменным ival и jval присваивается значение 0. Следующий пример неправилен, потому что типы pval и ival различны, и неявное преобразование типов

```
int main ()
{
    int ival; int *pval;
    ival = pval = 0; // ошибка: разные типы
```

невозможно. Отметим, что 0 является допустимым значением для обеих переменных:

```
    // ...
}
```

Верен или нет приведенный ниже пример, мы сказать не можем, , поскольку определение

```
int main()
{
    // ...
    int ival = jval = 0; // верно или нет?
    // ...
```

jval в нем отсутствует:

```
}
```

Это правильно только в том случае, если переменная jval определена в программе ранее и имеет тип, приводимый к int. Обратите внимание: в этом случае мы присваиваем 0 значение jval и инициализируем ival. Для того чтобы инициализировать нулем обе

```
int main()
{
    // правильно: определение и инициализация
    int ival = 0, jval = 0;
    // ...
```

переменные, мы должны написать:

```
}
```

В практике программирования часты случаи, когда к объекту применяется некоторая операция, а результат этой операции присваивается тому же объекту. Например:

```
int arraySum( int ia[], int sz )
{
    int sum = 0;
    for ( int i = 0; i < sz; ++i )
        sum = sum + ia[ i ];
    return sum;
}
```

Для более компактной записи C и C++ предоставляют составные операции присваивания. С использованием такого оператора данный пример можно переписать следующим образом:

```
int arraySum( int ia[], int sz )
{
    int sum = 0;
    for ( int i =0; i < sz; ++i )
        // эквивалентно: sum = sum + ia[ i ];
        sum += ia[ i ];
    return sum;
}
```

Общий синтаксис составного оператора присваивания таков:

```
a op= b;
```

где op= является одним из десяти операторов:

+=	-=	*=	/=	%=
<<=	>>=	&=	^=	|=

Запись a op= b в точности эквивалентна записи a = a op b.

Упражнение 4.6

```
int main() {
    float fval;
    int ival;
    int *pi;

    fval = ival = pi = 0;
}
```

Найдите ошибку в данном примере. Исправьте запись.

Упражнение 4.7

Следующие выражения синтаксически правильны, однако скорее всего работают не так, как предполагал программист. Почему? Как их изменить?

```
(a) if ( ptr = retrieve_pointer() != 0 )
(b) if ( ival = 1024 )

(c) ival += ival + 1;
```

4.5. Операции инкремента и декремента

Операции инкремента (++) и декремента (--) дают возможность компактной и удобной записи для изменения значения переменной на единицу. Чаще всего они используются при работе с массивами и коллекциями – для изменения величины индекса, указателя

```
#include <vector>
#include <cassert>

int main()
{
    int ia[10] = {0,1,2,3,4,5,6,7,8,9};
    vector<int> ivec( 10 );

    int ix_vec = 0, ix_ia = 9;
    while ( ix_vec < 10 )
        ivec[ ix_vec++ ] = ia[ ix_ia-- ];

    int *pia = &ia[9];
    vector<int>::iterator iter = ivec.begin();

    while ( iter != ivec.end() )
        assert( *iter++ == *pia-- );
```

или итератора:

```
}
```

Выражение

```
ix_vec++
```

является *постфиксной* формой оператора инкремента. Значение переменной ix_vec увеличивается *после того*, как ее текущее значение употреблено в качестве индекса. Например, на первой итерации цикла значение ix_vec равно 0. Именно это значение применяется как индекс массива ivec, после чего ix_vec увеличивается и становится равным 1, однако новое значение используется только на следующей итерации. Постфиксная форма операции декремента работает точно так же: текущее значение ix_ia берется в качестве индекса для ia, затем ix_ia уменьшается на 1.

Существует и *префиксная* форма этих операторов. При использовании такой формы текущее значение сначала уменьшается или увеличивается, а затем используется новое

```
// неверно: ошибки с границами индексов в
// обоих случаях
int ix_vec = 0, ix_ia = 9;
while ( ix_vec < 10 )
```

значение. Если мы пишем:

```
    ivec[ ++ix_vec ] = ia[ --ix_ia ];
```

значение ix_vec увеличивается на единицу и становится равным 1 до первого использования в качестве индекса. Аналогично ix_ia получает значение 8 при первом использовании. Для того чтобы наша программа работала правильно, мы должны

```
// правильно
int ix_vec = -1, ix_ia = 8;
while ( ix_vec < 10 )
```

скорректировать начальные значения переменных ix_ivec и ix_ia:

```
    ivec[ ++ix_vec ] = ia[ --ix_ia ];
```

В качестве последнего примера рассмотрим понятие *стека*. Это фундаментальная абстракция компьютерного мира, позволяющая помещать и извлекать элементы в последовательности *LIFO* (last in, fist out – последним вошел, первым вышел). Стек реализует две основные операции – поместить (push) и извлечь (pop).

Текущий свободный элемент называют вершиной стека. Операция push присваивает этому элементу новое значение , после чего вершина смещается вверх (становится на 1 больше). Пусть наш стек использует для хранения элементов вектор. Какую из форм операции увеличения следует применить? Сначала мы используем текущее значение, потом увеличиваем его. Это постфиксная форма:

```
stack[ top++ ] = value;
```

Что делает операция pop? Уменьшает значение вершины (текущая вершина показывает на пустой элемент), затем извлекает значение. Это префиксная форма операции уменьшения:

```
int value = stack[ --top ];
```

(Реализация класса stack приведена в конце этой главы. Стандартный класс stack рассматривается в разделе 6.16.)

Упражнение 4.8

Как вы думаете, почему язык программирования получил название C++, а не ++C?

4.6. Операции с комплексными числами

Класс комплексных чисел стандартной библиотеки C++ представляет собой хороший пример использования объектной модели. Благодаря перегруженным арифметическим операциям объекты этого класса используются так, как будто они принадлежат одному из встроенных типов данных. Более того, в подобных операциях могут одновременно принимать участие и переменные встроенного арифметического типа, и комплексные числа. (Отметим, что здесь мы не рассматриваем общие вопросы математики комплексных чисел. См. [PERSON68] или любую книгу по математике.) Например, можно написать:

```
#inc1ude <complex>

comp1ex< double > a;
comp1ex< double > b;

// ...
```

```
complex< double > c = a * b + a / b;
```

Комплексные и арифметические типы разрешается смешивать в одном выражении:

```
complex< double > complex_obj = a + 3.14159;
```

Аналогично комплексные числа инициализируются арифметическим типом, и им может

```
double dval = 3.14159;
```

быть присвоено такое значение:

```
complex_obj = dval;
```

```
int ival = 3;
```

Или

```
complex_obj = ival;
```

Однако обратное неверно. Например, следующее выражение вызовет ошибку

```
// ошибка: нет неявного преобразования
// в арифметический тип
```

компиляции:

```
double dval = complex_obj;
```

Нужно явно указать, какую часть комплексного числа – вещественную или мнимую – мы хотим присвоить обычному числу. Класс комплексных чисел имеет две функции, возвращающих соответственно вещественную и мнимую части. Мы можем обращаться к

```
double re = complex_obj.real();
```

ним, используя синтаксис доступа к членам класса:

```
double im = complex_obj.imag();
```

```
double re = real(complex_obj);
```

или эквивалентный синтаксис вызова функции:

```
double im = imag(complex_obj);
```

Класс комплексных чисел поддерживает четыре составных оператора присваивания: +=, -=, *= и /=. Таким образом,

```
complex_obj += second_complex_obj;
```

Поддерживается и ввод/вывод комплексных чисел. Оператор вывода печатает вещественную и мнимую части через запятую, в круглых скобках. Например, результат

```
complex< double > complex0( 3.14159, -2.171 );
comp1ex< double > complex1( complexO.real() );
```

выполнения операторов вывода

```
cout << complexO << " " << complex1 << endl;
```

выглядит так:

```
( 3.14159, -2.171 ) ( 3.14159, 0.0 )
```

```
// допустимые форматы для ввода комплексного числа
// 3.14159        ==> comp1ex( 3.14159 );
// ( 3.14159 )    ==> comp1ex( 3.14159 );
// ( 3.14, -1.0 ) ==> comp1ex( 3.14, -1.0 );

// может быть считано как
// cin >> a >> b >> c
// где a, b, c - комплексные числа
```

Оператор ввода понимает любой из следующих форматов:

```
3.14159 ( 3.14159 ) ( 3.14, -1.0 )
```

Кроме этих операций, класс комплексных чисел имеет следующие функции-члены: sqrt(), abs(), polar(), sin(), cos(), tan(), exp(), log(), log10() и pow().

Упражнение 4.9

Реализация стандартной библиотеки C++, доступная нам в момент написания книги, не поддерживает составных операций присваивания, если правый операнд не является комплексным числом. Например, подобная запись недопустима:

```
complex_obj += 1;
```

(Хотя согласно стандарту C++ такое выражение должно быть корректно, производители часто не успевают за стандартом.) Мы можем определить свой собственный оператор для реализации такой операции. Вот вариант функции, реализующий оператор сложения для complex<double>:

```
#include <complex>
inline complex<double>&
operator+=( complex<double> &cval, double dval )
{
    return cval += complex<double>( dval );
}
```

(Это пример перегрузки оператора для определенного типа данных, детально рассмотренной в главе 15.)

Используя этот пример, реализуйте три других составных оператора присваивания для типа `complex<double>`. Добавьте свою реализацию к программе, приведенной ниже, и

```
#include <iostream>
#include <complex>

// определения операций...

int main() {
    complex< double > cval ( 4.0, 1.0 );

    cout << cval << endl;
    cval += 1;
    cout << cval << endl;
    cval -= 1;
    cout << cval << endl;
    cval *= 2;
    cout << cval << endl;
    cout /= 2;
    cout << cval << endl;
}
```

запустите ее для проверки.

Упражнение 4.10

Стандарт C++ не специфицирует реализацию операций инкремента и декремента для комплексного числа. Однако их семантика вполне понятна: если уж мы можем написать:

```
cval += 1;
```

что означает увеличение на `1` вещественной части `cval`, то и операция инкремента выглядела бы вполне законно. Реализуйте эти операции для типа `complex<double>` и выполните следующую программу:

```
#include <iostream>
#include <complex>

// определения операций...

int main() {
    complex< double > cval( 4.0, 1.0 );

    cout << cval << endl;
    ++cva1;
    cout << cval << endl;

}
```

4.7. Условное выражение

Условное выражение, или *оператор выбора*, предоставляет возможность более

```
bool is_equal;
if (!strcmp(str1,str2)) is_equal = true;
```

компактной записи текстов, включающих инструкцию `if-else`. Например, вместо:

```
else                    is_equal = false;
```

можно употребить более компактную запись:

```
bool is_equa1 = !strcmp( strl, str2 ) ? true : false;
```

Условный оператор имеет следующий синтаксис:

```
expr11 ? expr2 : expr3;
```

Вычисляется выражение `expr1`. Если его значением является `true`, оценивается `expr2`,

```
int min( int ia, int ib )
```

если `false`, то `expr3`. Данный фрагмент кода:

```
{ return ( ia < ib ) ? ia : ib; }
```

```
int min(int ia, int ib) {
  if (ia < ib)
    return ia;
  else
    return ib;
```

эквивалентен

```
}
```

Приведенная ниже программа иллюстрирует использование условного оператора:

```
#include <iostream>

int main()
{
   int i = 10, j = 20, k = 30;
   cout << "Большим из "
        << i << " и " << j << " является "
        << ( i > j ? i : j ) << end1;

   cout << "Значение " << i
        << ( i % 2 ? " нечетно." : " четно." )
        << endl;

/* условный оператор может быть вложенным,
 * но глубокая вложенность трудна для восприятия.
 * В данном примере max получает значение
 * максимальной из трех величин
 */
    int max = ( (i > j)
        ? (( i > k) ? i : k)
        : ( j > k ) ? j : k);

    cout << "Большим из "
         << i << ", " << j << " и " << k
         << " является " << max << endl;

}
```

Результатом работы программы будет:

```
Большим из 10 и 20 является 20
Значение 10 четно.
```

4.8. Оператор sizeof

Оператор `sizeof` возвращает размер в байтах объекта или типа данных. Синтаксис его

```
sizeof ( type name );
sizeof ( object );
```

таков:

```
sizeof object;
```

Результат имеет специальный тип `size_t`, который определен как `typedef` в заголовочном файле `cstddef`. Вот пример использования обеих форм оператора `sizeof`:

```
#include <cstddef>

int ia[] = { 0, 1, 2 };

// sizeof возвращает размер всего массива
size_t array_size = sizeof ia;

// sizeof возвращает размер типа int
size_t element_size = array_size / sizeof( int );
```

Применение sizeof к массиву дает количество байтов, занимаемых массивом, а не количество его элементов и не размер в байтах каждого из них. Так, например, в системах, где int хранится в 4 байтах, значением array_size будет 12. Применение

```
int *pi = new int[ 3 ];
```

sizeof к указателю дает размер самого указателя, а не объекта, на который он указывает:

```
size_t pointer_size = sizeof ( pi );
```

Здесь значением pointer_size будет память под указатель в байтах (4 в 32-битных системах), а не массива ia.

Вот пример программы, использующей оператор sizeof:

```
#include <string>
#include <iostream>
#include <cstddef>

int main() {
    size_t ia;

    ia = sizeof( ia ); //     правильно
    ia = sizeof ia;    //     правильно

    // ia = sizeof int; // ошибка
    ia = sizeof( int ); // правильно

    int *pi = new int[ 12 ];
    cout << "pi: " << sizeof( pi )
         << " *pi: " << sizeof( pi )
         << endl;

    // sizeof строки не зависит от
    // ее реальной длины
    string stl( "foobar" );
    string st2( "a mighty oak" );

    string *ps = &stl;

    cout << " st1: " << sizeof( st1 )
         << " st2: " << sizeof( st2 )
         << " ps: sizeof( ps )
         << " *ps: " << sizeof( *ps )
         << endl;

    cout << "short :\t"    << sizeof(short)    << endl;
    cout << "shorf" :\t"   << sizeof(short*)   << endl;
    cout << "short& :\t"   << sizeof(short&)   << endl;
    cout << "short[3] :\t" << sizeof(short[3]) << endl;
}
```

Результатом работы программы будет:

```
pi: 4 *pi: 4
st1: 12 st2: 12 ps: 4 *ps:12
short : 2
short* : 4
short& : 2
short[3] : 6
```

Из данного примера видно, что применение `sizeof` к указателю позволяет узнать размер памяти, необходимой для хранения адреса. Если же аргументом `sizeof` является ссылка, мы получим размер связанного с ней объекта.

```
// char_size == 1
```

Гарантируется, что в любой реализации C++ размер типа `char` равен 1.

```
size_t char_size = sizeof( char );
```

Значение оператора sizeof вычисляется во время компиляции и считается константой. Оно может быть использовано везде, где требуется константное значение, в том числе в

```
// правильно: константное выражение
```

качестве размера встроенного массива. Например:

```
int array[ sizeof( some_type_T )];
```

4.9. Операторы new и delete

Каждая программа во время работы получает определенное количество памяти, которую можно использовать. Такое выделение памяти под объекты во время выполнения называется *динамическим*, а сама память выделяется из *хипа* (heap). (Мы уже касались вопроса о динамическом выделении памяти в главе 1.) Напомним, что выделение памяти объекту производится с помощью оператора new, возвращающего указатель на вновь созданный объект того типа, который был ему задан. Например:

```
int *pi = new int;
```

размещает объект типа int в памяти и инициализирует указатель pi адресом этого объекта. Сам объект в таком случае не инициализируется, но это легко изменить:

```
int *pi = new int( 1024 );
```

Можно динамически выделить память под массив:

```
int *pia = new int[ 10 ];
```

Такая инструкция размещает в памяти массив встроенного типа из десяти элементов типа int. Для подобного массива нельзя задать список начальных значений его элементов при динамическом размещении. (Однако если размещается массив объектов типа класса, то для каждого из элементов вызывается конструктор по умолчанию.) Например:

```
string *ps = new string;
```

размещает в памяти один объект типа string, инициализирует ps его адресом и вызывает конструктор по умолчанию для вновь созданного объекта типа string. Аналогично

```
string *psa = new string[10];
```

размещает в памяти массив из десяти элементов типа string, инициализирует psa его адресом и вызывает конструктор по умолчанию для каждого элемента массива.

Объекты, размещаемые в памяти с помощью оператора new, не имеют собственного имени. Вместо этого возвращается указатель на безымянный объект, и все действия с этим объектом производятся посредством косвенной адресации.

После использования объекта, созданного таким образом, мы должны явно освободить память, применив оператор delete к указателю на этот объект. (Попытка применить

оператор delete к указателю, не содержащему адрес объекта, полученного описанным способом, вызовет ошибку времени выполнения.) Например:

```
delete pi;
```

освобождает память, на которую указывает объект типа int, на который указывает pi. Аналогично

```
delete ps;
```

освобождает память, на которую указывает объект класса string, адрес которого содержится в ps. Перед уничтожением этого объекта вызывается деструктор. Выражение

```
delete [] pia;
```

освобождает память, отведенную под массив pia. При выполнении такой операции необходимо придерживаться указанного синтаксиса.

(Об операциях new и delete мы еще поговорим в главе 8.)

Упражнение 4.11

Какие из следующих выражений ошибочны?

(a) `vector<string> svec( 10 );`

(b) `vector<string> *pvecl = new vector<string>(10);`

(c) `vector<string> **pvec2 = new vector<string>[10];`

(d) `vector<string> *pvl = &svec;`

(e) `vector<string> *pv2 = pvecl;`

(f) `delete svec;`

(g) `delete pvecl;`

(h) `delete [] pvec2;`

(i) `delete pvl;`

(j) `delete pv2;`

4.10. Оператор “запятая”

Одно выражение может состоять из набора подвыражений, разделенных запятыми; такие подвыражения вычисляются слева направо. Конечным результатом будет результат самого правого из них. В следующем примере каждое из подвыражений условного оператора представляет собой список. Результатом первого подвыражения условного оператора является ix, второго – 0.

```
int main()
{
    // примеры оператора "запятая"
    // переменные ia, sz и index определены в другом месте ...
    int ival = (ia != 0)
               ? ix=get_va1ue(), ia[index]=ix
               : ia=new int[sz], ia[index]=0;
    // ...
}
```

4.11. Побитовые операторы

Таблица 4.3. Побитовые операторы

Символ операции	Значение	Использование
~	Побитовое НЕ	~expr
<<	Сдвиг влево	expr1 << expr2
>>	Сдвиг вправо	expr1 >> expr2
&	Побитовое И	expr1 & expr2
^	Побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ	expr1 ^ expr2
|	Побитовое ИЛИ	expr1 | expr2
&=	Побитовое И с присваиванием	expr1 &= expr2
^=	Побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ с присваиванием	expr1 ^= expr2
|=	Побитовое ИЛИ с присваиванием	expr1 |= expr2
<<=	Сдвиг влево с присваиванием	expr1 <<= expr2
>>=	Сдвиг вправо с присваиванием	expr1 >>= expr2

Побитовые операции рассматривают операнды как упорядоченные наборы битов, каждый бит может иметь одно из двух значений – `0` или `1`. Такие операции позволяют программисту манипулировать значениями отдельных битов. Объект, содержащий набор битов, иногда называют *битовым вектором*. Он позволяет компактно хранить набор *флагов* – переменных, принимающих значение “да” “нет”. Например, компиляторы зачастую помещают в битовые векторы спецификаторы типов, такие, как `const` и `volatile`. Библиотека `iostream` использует эти векторы для хранения состояния формата вывода.

Как мы видели, в C++ существуют два способа работы со строками: использование C-строк и объектов типа `string` стандартной библиотеки – и два подхода к массивам: массивы встроенного типа и объект `vector`. При работе с битовыми векторами также

можно применять подход, заимствованный из C, – использовать для представления такого вектора объект встроенного целого типа, обычно `unsigned int`, или класс `bitset` стандартной библиотеки C++. Этот класс инкапсулирует семантику вектора, предоставляя операции для манипулирования отдельными битами. Кроме того, он позволяет ответить на вопросы типа: есть ли "взведенные" биты (со значением 1) в векторе? Сколько битов "взведено"?

В общем случае предпочтительнее пользоваться классом `bitset`, однако, понимание работы с битовыми векторами на уровне встроенных типов данных очень полезно. В этом разделе мы рассмотрим применение встроенных типов для представления битовых векторов, а в следующем – класс `bitset`.

При использовании встроенных типов для представления битовых векторов можно пользоваться как знаковыми, так и беззнаковыми целыми типами, но мы настоятельно советуем пользоваться беззнаковыми: поведение побитовых операторов со знаковыми типами может различаться в разных реализациях компиляторов.

Побитовое НЕ (`~`) меняет значение каждого бита операнда. Бит, установленный в `1`, меняет значение на `0` и наоборот.

Операторы сдвига (`<<`, `>>`) сдвигают биты в левом операнде на указанное правым операндом количество позиций. "Выталкиваемые наружу" биты пропадают, освобождающиеся биты (справа для сдвига влево, слева для сдвига вправо) заполняются нулями. Однако нужно иметь в виду, что для сдвига вправо заполнение левых битов нулями гарантируется только для беззнакового операнда, для знакового в некоторых реализациях возможно заполнение значением знакового (самого левого) бита.

Побитовое И (`&`) применяет операцию И ко всем битам своих операндов. Каждый бит левого операнда сравнивается с битом правого, находящимся в той же позиции. Если оба бита равны `1`, то бит в данной позиции получает значение `1`, в любом другом случае – `0`. (Побитовое И (`&`) не надо путать с логическим И (`&&`),но, к сожалению, каждый программист хоть раз в жизни совершал подобную ошибку.)

Побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ (`^`) сравнивает биты операндов. Соответствующий бит результата равен `1`, если операнды различны (один равен `0`, а другой `1`). Если же оба операнда равны, результата равен `0`.

Побитовое ИЛИ (`|`) применяет операцию логического сложения к каждому биту операндов. Бит в позиции результата получает значение `1`, если хотя бы один из соответствующих битов операндов равен `1`, и `0`, если биты обоих операндов равны `0`. (Побитовое ИЛИ не нужно смешивать с логическим ИЛИ.)

Рассмотрим простой пример. Пусть у нас есть класс из 30 студентов. Каждую неделю преподаватель проводит зачет, результат которого – сдал/не сдал. Итоги можно представить в виде битового вектора. (Заметим, что нумерация битов начинается с нуля, первый бит на самом деле является вторым по счету. Однако для удобства мы не будем использовать нулевой бит; таким образом, студенту номер 1 соответствует бит номер 1. В конце концов, наш преподаватель – не специалист в области программирования.)

```
unsigned int quiz1 = 0;
```

Нам нужно иметь возможность менять значение каждого бита и проверять это значение. Предположим, студент 27 сдал зачет. Бит 27 необходимо выставить в 1, не меняя значения других битов. Это можно сделать за два шага. Сначала нужно начать с числа, содержащего `1` в 27-м бите и `0` в остальных. Для этого используем операцию сдвига:

```
1 << 27;
```

Применив побитовую операцию ИЛИ к переменной `quiz1` и нашей константе, получим нужный результат: значение 27-й бита станет равным значение `1`, а другие биты останутся неизменными.

```
quiz1 |= 1<<27;
```

Теперь представим себе, что преподаватель перепроверил результаты теста и выяснил, что студент 27 зачет не сдал. Теперь нужно присвоить нуль 27-му биту, не трогая остальных. Сначала применим побитовое НЕ к предыдущей константе и получим число, в котором все биты, кроме 27-го, равны `1`:

```
~(1<<27 );
```

Теперь побитово умножим (И) эту константу на `quiz1` и получим нужный результат: `0` в 27-м бите и неизменные значения остальных.

```
quiz1 &= ~(1<<27);
```

Как проверить значение того же 27-го бита? Побитовое И дает `true`, если 27-й бит равен `1`, и `false`, если `0`:

```
bool hasPassed = quiz1 & (1<<27);
```

При использовании побитовых операций подобным образом очень легко допустить ошибку. Поэтому чаще всего такие операции инкапсулируются в макросы препроцессора

```
inline boo1 bit_on (unsigned int ui, int   pos)
{
    return u1 & ( 1 << pos );
```

или встроенные функции:

```
}
```

```
enum students { Danny = 1, Jeffrey, Ethan, Zev, Ebie, // ...
                AnnaP = 26, AnnaL = 27 };
const int student_size = 27;

// наш битовый вектор начинается с 1
boo1 has_passed_quiz[ student_size+l ];
for ( int index = 1; index <= student_size; ++-index )
```

Вот пример использования:

```
    has_passed_quiz[ index ] = bit_on( quiz1, index );
```

Раз уж мы начали инкапсулировать действия с битовым вектором в функции, следующим шагом нужно создать класс. Стандартная библиотека C++ включает такой класс `bitset`, его использование описано ниже.

Упражнение 4.12

Даны два целых числа:

```
unsigned int ui1 = 3, ui2 = 7;
```

Каков результат следующих выражений?

(a) `ui1 & ui2` (c) `uil | ui2`

(b) `ui1 && ui2` (d) `uil || ui2`

Упражнение 4.13

Используя пример функции `bit_on()`, создайте функции `bit_turn_on()` (выставляет бит в `1`), `bit_turn_off()` (сбрасывает бит в `0`), `flip_bit()` (меняет значение на противоположное) и `bit_off()` (возвращает `true`, если бит равен `0`). Напишите программу, использующую ваши функции.

Упражнение 4.14

В чем недостаток функций из предыдущего упражнения, использующих тип `unsigned int`? Их реализацию можно улучшить, используя определение типа с помощью `typedef` или механизм функций-шаблонов. Перепишите функцию `bit_on()`,применив сначала `typedef`, а затем механизм шаблонов.

4.12. Класс bitset

Таблица 4.4. Операции с классом `bitset`

Операция	Значение	Использование
`test(pos)`	Бит `pos` равен `1`?	`a.test(4)`
`any()`	Хотя бы один бит равен `1`?	`a.any()`
`none()`	Ни один бит не равен `1`?	`a.none()`
`count()`	Количество битов, равных `1`	`a.count()`
`size()`	Общее количество битов	`a.size()`
`[pos]`	Доступ к биту `pos`	`a[4]`
`flip()`	Изменить значения всех	`a.flip()`
`flip(pos)`	Изменить значение бита `pos`	`a.flip(4)`
`set()`	Выставить все биты в `1`	`a.set()`
`set(pos)`	Выставить бит `pos` в `1`	`a.set(4)`
`reset()`	Выставить все биты в `0`	`a.reset()`
`reset(pos)`	Выставить бит `pos` в `0`	`a.reset(4)`

Как мы уже говорили, необходимость создавать сложные выражения для манипуляции битовыми векторами затрудняет использование встроенных типов данных. Класс `bitset` упрощает работу с битовым вектором. Вот какое выражение нам приходилось писать в предыдущем разделе для того, чтобы “взвести” 27-й бит:

```
quiz1 |= 1<<27;
```

При использовании bitset то же самое мы можем сделать двумя способами:

```
quiz1[27] = 1;
```

или

```
quiz1.set(27);
```

(В нашем примере мы не используем нулевой бит, чтобы сохранить “естественную” нумерацию. На самом деле, нумерация битов начинается с 0.)

Для использования класса bitset необходимо включить заголовочный файл:

```
#include <bitset>
```

Объект типа bitset может быть объявлен тремя способами. В определении по умолчанию мы просто указываем размер битового вектора:

```
bitset<32> bitvec;
```

Это определение задает объект bitset, содержащий 32 бита с номерами от 0 до 31. Все биты инициализируются нулем. С помощью функции any() можно проверить, есть ли в векторе единичные биты. Эта функция возвращает true, если хотя бы один бит отличен от нуля. Например:

```
bool is_set = bitvec.any();
```

Переменная is_set получит значение false, так как объект bitset по умолчанию инициализируется нулями. Парная функция none() возвращает true, если все биты равны нулю:

```
bool is_not_set = bitvec.none();
```

Изменить значение отдельного бита можно двумя способами: воспользовавшись функциями set() и reset() или индексом. Так, следующий цикл выставляет в 1

```
for ( int index=0; index<32; ++index )
    if ( index % 2 == 0 )
```

каждый четный бит:

```
        bitvec[ index ] = 1;
```

Аналогично существует два способа проверки значений каждого бита – с помощью функции test() и с помощью индекса. Функция () возвращает true, если

```
if ( bitvec.test( 0 ))
```

соответствующий бит равен 1, и false в противном случае. Например:

```
    // присваивание bitvec[0]=1 сработало!;
```

```
cout << "bitvec: включенные биты:\n\t";
for ( int index = 0; index < 32; ++-index )
    if ( bitvec[ index ] )
        cout << index << " ";
```

Значения битов с помощью индекса проверяются таким образом:

```
cout << endl;
```

```
bitvec.reset(0);
```

Следующая пара операторов демонстрирует сброс первого бита двумя способами:

```
bitvec[0] = 0;
```

Функции `set()` и `reset()` могут применяться ко всему битовому вектору в целом. В

```
// сброс всех битов
bitvec.reset();
if (bitvec.none() != true)
// что-то не сработало
// установить в 1 все биты вектора bitvec
if ( bitvec.any() != true )
```

этом случае они должны быть вызваны без параметра. Например:

```
    // что-то опять не сработало
```

```
bitvec.f1ip( 0 );  // меняет значение первого бита
bitvec[0].flip();  // тоже меняет значение первого бита
```

Функция `flip()` меняет значение отдельного бита или всего битового вектора:

```
bitvec.flip();     // меняет значения всех битов
```

Существуют еще два способа определить объект типа `bitset`. Оба они дают возможность проинициализировать объект определенным набором нулей и единиц. Первый способ – явно задать целое беззнаковое число как аргумент конструктору. Начальные N позиций битового вектора получат значения соответствующих двоичных разрядов аргумента. Например:

```
bitset< 32 > bitvec2( 0xffff );
```

инициализирует `bitvec2` следующим набором значений:

```
00000000000000001111111111111111
```

В результате определения

```
bitset< 32 > bitvec3( 012 );
```

у bitvec3 окажутся ненулевыми биты на местах 1 и 3:

```
00000000000000000000000000001010
```

В качестве аргумента конструктору может быть передано и строковое значение, состоящее из нулей и единиц. Например, следующее определение инициализирует

```
// эквивалентно bitvec3
string bitva1( "1010" );
```

bitvec4 тем же набором значений, что и bitvec3:

```
bitset< 32 > bitvec4( bitval );
```

Можно также указать диапазон символов строки, выступающих как начальные значения

```
// подстрока с шестой позиции длиной 4: 1010
string bitval ( "11111101011000110101010101" );
```

для битового вектора. Например:

```
bitset< 32 > bitvec5( bitval, 6, 4 );
```

Мы получаем то же значение, что и для bitvec3 и bitvec4. Если опустить третий

```
// подстрока с шестой позиции до конца строки: 1010101
string bitva1( "11111101011000110101010101" );
```

параметр, подстрока берется до конца исходной строки:

```
bitset< 32 > bitvec6( bitval, 6 );
```

Класс bitset предоставляет две функции-члена для преобразования объекта bitset в другой тип. Для трансформации в строку, состоящую из символов нулей и единиц, служит функция to_string():

```
string bitva1( bitvec3.to_string() );
```

Вторая функция, to_long(), преобразует битовый вектор в его целочисленное представление в виде unsigned long, если, конечно, оно помещается в unsigned long. Это видоизменение особенно полезно, если мы хотим передать битовый вектор функции на C или C++, не пользующейся стандартной библиотекой.

К объектам типа bitset можно применять побитовые операции. Например:

```
bitset<32> bitvec7 = bitvec2 & bitvec3;
```

Объект bitvec7 инициализируется результатом побитового И двух битовых векторов bitvec2 и bitvec3.

```
bitset<32> bitvec8 = bitvec2 | bitvec3;
```

Здесь bitvec8 инициализируется результатом побитового ИЛИ векторов bitvec2 и bitvec3. Точно так же поддерживаются и составные операции присваивания и сдвига.

Упражнение 4.15

Допущены ли ошибки в приведенных определениях битовых векторов?

(a) `bitset<64> bitvec(32);`

(b) `bitset<32> bv( 1010101 );`

(c) `string bstr; cin >> bstr; bitset<8>bv( bstr );`

(d) `bitset<32> bv; bitset<16> bvl6( bv );`

Упражнение 4.16

```
extern void bitstring(const char*);
bool bit_on (unsigned long, int);
bitset<32> bitvec;

(a) bitsting( bitvec.to_string().c_str() );
(b) if ( bit_on( bitvec.to_1ong(), 64 )) ...
```

Допущены ли ошибки в следующих операциях с битовыми векторами?

```
(c) bitvec.f1ip( bitvec.count() );
```

Упражнение 4.17

Дана последовательность: 1,2,3,5,8,13,21. Каким образом можно инициализировать объект bitset<32> для ее представления? Как присвоить значения для представления этой последовательности пустому битовому вектору? Напишите вариант инициализации и вариант с присваиванием значения каждому биту.

4.13. Приоритеты

Приоритеты операций задают последовательность вычислений в сложном выражении. Например, какое значение получит ival?

```
int ival = 6 + 3 * 4 / 2 + 2;
```

Если вычислять операции слева направо, получится 20. Среди других возможных результатов будут 9, 14 и 36. Правильный ответ: 14.

В C++ умножение и деление имеют более высокий приоритет, чем сложение, поэтому они будут вычислены раньше. Их собственные приоритеты равны, поэтому умножение и деление будут вычисляться слева направо. Таким образом, порядок вычисления данного выражения таков:

```
1. 3 * 4 => 12
2. 12 / 2 => 6
3. 6 + 6 => 12
4. 12 + 2 => 14
```

Следующая конструкция ведет себя не так, как можно было бы ожидать. Приоритет операции присваивания меньше, чем операции сравнения:

```
while ( ch = nextChar() != '\n' )
```

Программист хотел присвоить переменной ch значение, а затем проверить, равно ли оно символу новой строки. Однако на самом деле выражение сначала сравнивает значение, полученное от nextChar(), с '\n', и результат – true или false – присваивает переменной ch.

Приоритеты операций можно изменить с помощью скобок. Выражения в скобках вычисляются в первую очередь. Например:

```
4 * 5 + 7 * 2 ==> 34
4 * ( 5 + 7 * 2 ) ==> 76
4 * ( (5 + 7) * 2 ) ==> 96
```

Вот как с помощью скобок исправить поведение предыдущего примера:

```
while ( (ch = nextChar()) != '\n' )
```

Операторы обладают и приоритетом, и ассоциативностью. Оператор присваивания правоассоциативен, поэтому вычисляется справа налево:

```
ival = jval = kva1 = lval
```

Сначала kval получает значение lval, затем jval – значение результата этого присваивания, и в конце концов ival получает значение jval.

Арифметические операции, наоборот, левоассоциативны. Следовательно, в выражении

```
ival + jval + kva1 + 1va1
```

сначала складываются ival и jval, потом к результату прибавляется kval, а затем и lval.

В таблице 4.4 приведен полный список операторов C++ в порядке уменьшения их приоритета. Операторы внутри одной секции таблицы имеют равные приоритеты. Все операторы некоторой секции имеют более высокий приоритет, чем операторы из секций, следующих за ней. Так, операции умножения и деления имеют одинаковый приоритет, и он выше приоритета любой из операций сравнения.

Упражнение 4.18

```
(a) ! ptr == ptr->next
(b) ~ uc ^ 0377 & ui << 4
```

Каков порядок вычисления следующих выражений? При ответе используйте таблицу 4.4.

```
(c) ch = buf[ bp++ ] != '\n'
```

Упражнение 4.19

Все три выражения из предыдущего упражнения вычисляются не в той последовательности, какую, по-видимому, хотел задать программист. Расставьте скобки так, чтобы реализовать его первоначальный замысел.

Упражнение 4.20

Следующие выражения вызывают ошибку компиляции из-за неправильно понятого

```
(a) int i = doSomething(), 0;
```

приоритета операций. Объясните, как их исправить, используя таблицу 4.4.

```
(b) cout << ival % 2 ? "odd" : "even";
```

Таблица 4.4. Приоритеты операций

Оператор	**Значение**	**Использование**
::	Глобальная область видимости	::name
::	Область видимости класса	class::name
::	Область видимости пространства имен	namespace::name
.	Доступ к члену	object.member
->	Доступ к члену по указателю	pointer->member
[]	Взятие индекса	variable[expr]
()	Вызов функции	name(expr_list)
()	Построение значения	type(expr_list)
++	постфиксный инкремент	lvalue++
--	постфиксный декремент	lvalue--
typeid	идентификатор типа	typeid(type)
typeid	идентификатор типа выражения	typeid(expr)
const_cast	преобразование типа	const_cast<type>(expr)
dynamic_cast	преобразование типа	dynamic_cast<type>(expr)
reinterpret_cast	приведение типа	reinterpret_cast<type> (expr)
static_cast	приведение типа	static_cast<type>(expr)
sizeof	размер объекта	sizeof expr
sizeof	размер типа	sizeof(type)
++	префиксный инкремент	++lvalue
--	префиксный декремент	--lvalue

~	побитовое НЕ	~expr
!	логическое НЕ	!expr
-	унарный минус	-expr
+	унарный плюс	+expr
*	разыменование	*expr
&	адрес	&expr
()	приведение типа	(type)expr
new	выделение памяти	new type
new	выделение памяти и инициализация	new type(exprlist)
new	выделение памяти	new (exprlist) type(exprlist)
new	выделение памяти под массив	все формы
delete	освобождение памяти	все формы
delete	освобождение памяти из-под массива	все формы
->*	доступ к члену классу по указателю	pointer-> *pointer_to_member
.*	доступ к члену класса по указателю	object.*pointer_to_member
*	умножение	expr * expr
/	деление	expr / expr
%	деление по модулю	expr % expr
+	сложение	expr + expr
-	вычитание	expr - expr
<<	сдвиг влево	expr << expr
>>	сдвиг вправо	expr >> expr
<	меньше	expr < expr
<=	меньше или равно	expr <= expr
>	больше	expr > expr
>=	больше или равно	expr >= expr
==	равно	expr == expr
!=	не равно	expr != expr
&	побитовое И	expr & expr
^	побитовое ИСКЛЮЧАЮЩЕЕ	expr ^ expr

	ИЛИ	
|	побитовое ИЛИ	expr | expr
&&	логическое И	expr && expr
||	логическое ИЛИ	expr || expr
?:	условный оператор	expr ? expr * expr
=	присваивание	l-значение = expr
=, *=, /=, %=, +=, -=, <<=, >>=, &=, |=, ^=	составное присваивание	l-значение += expr и т.д.
throw	возбуждение исключения	throw expr
,	запятая	expr, expr

4.14. Преобразования типов

```
int ival = 0;

// обычно компилируется с предупреждением
```

Представим себе следующий оператор присваивания:

```
ival = 3.541 + 3;
```

В результате ival получит значение 6. Вот что происходит: мы складываем литералы разных типов – 3.541 типа double и 3 типа int. C++ не может непосредственно сложить подобные операнды, сначала ему нужно привести их к одному типу. Для этого существуют правила *преобразования арифметических типов*. Общий принцип таков: перейти от операнда меньшего типа к большему, чтобы не потерять точность вычислений.

В нашем случае целое значение 3 трансформируется в тип double, и только после этого производится сложение. Такое преобразование выполняется независимо от желания программиста, поэтому оно получило название *неявного преобразования типов*.

Результат сложения двух чисел типа double тоже имеет тип double. Значение равно 6.541. Теперь его нужно присвоить переменной ival. Типы переменной и результата 6.541 не совпадают, следовательно, тип этого значения приводится к типу переменной слева от знака равенства. В нашем случае это int. Преобразование double в int производится автоматически, отбрасыванием дробной части (а не округлением). Таким образом, 6.541 превращается в 6, и этот результат присваивается переменной ival. Поскольку при таком преобразовании может быть потеряна точность, большинство компиляторов выдают предупреждение.

Так как компилятор не округляет числа при преобразовании double в int, при

```
double dva1 = 8.6;
int iva1 = 5;
```

необходимости мы должны позаботиться об этом сами. Например:

```
ival += dva1 + 0.5; // преобразование с округлением


// инструкция компилятору привести double к int
```

При желании мы можем произвести *явное преобразование типов*:

```
ival = static_cast< int >( 3.541 ) + 3;
```

В этом примере мы явно даем указание компилятору привести величину `3.541` к типу `int`, а не следовать правилам по умолчанию.

В этом разделе мы детально обсудим вопросы и неявного (как в первом примере), и явного преобразования типов (как во втором).

4.14.1. Неявное преобразование типов

Язык определяет набор стандартных преобразований между объектами встроенного типа, неявно выполняющихся компилятором в следующих случаях:

- арифметическое выражение с операндами разных типов: все операнды приводятся к наибольшему типу из встретившихся. Это называется

```
int   ival = 3;
double dva1 = 3.14159;
// ival преобразуется в double: 3.0
```

  арифметическим преобразованием. Например:

```
ival + dva1;
```

- присваивание значения выражения одного типа объекту другого типа. В этом случае результирующим является тип объекта, которому значение присваивается. Так, в первом примере литерал `0` типа `int` присваивается указателю типа `int*`,

```
// 0 преобразуется в нулевой указатель типа int*
int *pi = 0;

// dva1 преобразуется в int: 3
```

  значением которого будет `0`. Во втором примере `double` преобразуется в `int`.

```
ivat = dva1;
```

- передача функции аргумента, тип которого отличается от типа соответствующего формального параметра. Тип фактического аргумента

```
extern double sqrt( double );

// 2 преобразуется в double: 2.0
```

  приводится к типу параметра:

```
cout << "Квадратный корень из 2: " << sqrt( 2 ) << endt;
```

- возврат из функции значения, тип которого не совпадает с типом возвращаемого результата, заданным в объявлении функции. Тип фактически

```
double difference( int ivati, int iva12 )
{
    // результат преобразуется в double
    return ivati - iva12;
```

возвращаемого значения приводится к объявленному. Например:

}

4.14.2. Арифметические преобразования типов

Арифметические преобразования приводят оба операнда бинарного арифметического выражения к одному типу, который и будет типом результата выражения. Два общих правила таковы:

- типы всегда приводятся к тому из типов, который способен обеспечить наибольший диапазон значений при наибольшей точности. Это помогает уменьшить потери точности при преобразовании;
- любое арифметическое выражение, включающее в себя целые операнды типов, меньших чем int, перед вычислением всегда преобразует их в int.
- Мы рассмотрим иерархию правил преобразований, начиная с наибольшего типа long double.

Если один из операндов имеет тип long double, второй приводится к этому же типу в любом случае. Например, в следующем выражении символьная константа 'a' трансформируется в long double (значение 97 для представления ASCII) и затем прибавляется к литералу того же типа: 3.14159L + 'a'.

Если в выражении нет операндов long double, но есть операнд double, все

```
int    iva1;
float fval;
double dval;

// fva1 и iva1 преобразуются к double перед сложением
```

преобразуется к этому типу. Например:

```
dval + fva1 + ival;
```

В том случае, если нет операндов типа double и long double, но есть операнд float,

```
char cvat;
int iva1;
float fva1;

// iva1 и cval преобразуются к float перед сложением
```

тип остальных операндов меняется на float:

```
cvat + fva1 + iva1;
```

Если у нас нет вещественных операндов , значит, все они представляют собой целые типы. Прежде чем определить тип результата, производится преобразование, называемое *приведением к целому*: все операнды с типом меньше, чем int, заменяются на int.

При приведении к целому типы char, signed char, unsigned char и short int преобразуются в int. Тип unsigned short int трансформируется в int, если этот тип достаточен для представления всего диапазона значений unsigned short int (обычно это происходит в системах, отводящих полслова под short и целое слово под int), в противном случае unsigned short int заменяется на unsigned int.

Тип wchar_t и перечисления приводятся к наименьшему целому типу, способному представить все их значения. Например, в перечислении

```
enum status { bad, ok };
```

значения элементов равны 0 и 1. Оба эти значения могут быть представлены типом char, значит char и станет типом внутреннего представления данного перечисления. Приведение к целому преобразует char в int.

```
char cval;
bool found;
enum mumble { ml, m2, m3 }    mval;

unsigned long ulong;
```

В следующем выражении

```
cval + ulong; ulong + found; mval + ulong;
```

перед определением типа результата cval, found и mval преобразуются в int.

После приведения к целому сравниваются получившиеся типы операндов. Если один из них имеет тип unsigned long, то остальные будут того же типа. В нашем примере все три объекта, прибавляемые к ulong, приводятся к типу unsigned long.

Если в выражении нет объектов unsigned long, но есть объекты типа long, тип

```
char cval;
long lval;

// cval и 1024 преобразуются в long перед сложением
```

остальных операндов меняется на long. Например:

```
cval + 1024 + lval;
```

Из этого правила есть одно исключение: преобразование unsigned int в long происходит только в том случае, если тип long способен вместить весь диапазон значений unsigned int. (Обычно это не так в 32-битных системах, где и long, и int представляются одним машинным словом.) Если же тип long не способен представить весь диапазон unsigned int, оба операнда приводятся к unsigned long.

В случае отсутствия операндов типов unsigned long и long, используется тип unsigned int. Если же нет операндов и этого типа, то к int.

Может быть, данное объяснение преобразований типов несколько смутило вас. Запомните основную идею: арифметическое преобразование типов ставит своей целью сохранить точность при вычислении. Это достигается приведением типов всех операндов к типу, способному вместить любое значение любого из присутствующих в выражении операндов.

4.14.3. Явное преобразование типов

Явное преобразование типов производится при помощи следующих операторов: static_cast, dynamic_cast, const_cast и reinterpret_cast. Заметим, что, хотя иногда явное преобразование необходимо, оно служит потенциальным источником ошибок, поскольку подавляет проверку типов, выполняемую компилятором. Давайте сначала посмотрим, зачем нужно такое преобразование.

Указатель на объект любого неконстантного типа может быть присвоен указателю типа void*, который используется в тех случаях, когда действительный тип объекта либо неизвестен, либо может меняться в ходе выполнения программы. Поэтому указатель

```
int iva1;
int *pi = 0;
char *pc = 0;
void *pv;

pv = pi; // правильно: неявное преобразование
pv = pc; // правильно: неявное преобразование

const int *pci = &iva1;
pv = pci; // ошибка: pv имеет тип, отличный от const void*;
```

void* иногда называют *универсальным* указателем. Например:

```
const void *pcv = pci; // правильно
```

Однако указатель void* не может быть разыменован непосредственно. Компилятор не знает типа объекта, адресуемого этим указателем. Но это известно программисту, который хочет преобразовать указатель void* в указатель определенного типа. C++ не

```
#inc1ude <cstring>
int ival = 1024;
void *pv;
int *pi = &iva1;
const char *pc = "a casting call";

void mumble() {
    pv = pi; // правильно: pv получает адрес ival
    pc = pv; // ошибка: нет стандартного преобразования

    char *pstr = new char[ str1en( pc )+1 ];
    strcpy( pstr, pc );
```

обеспечивает подобного автоматического преобразования:

```
}
```

Компилятор выдает сообщение об ошибке, так как в данном случае указатель pv содержит адрес целого числа ival, и именно этот адрес пытаются присвоить указателю на строку. Если бы такая программа была допущена до выполнения, то вызов функции strcpy(), которая ожидает на входе строку символов с нулем в конце, скорее всего привел бы к краху, потому что вместо этого strcpy() получает указатель на целое число. Подобные ошибки довольно просто не заметить, именно поэтому C++ запрещает неявное преобразование указателя на void в указатель на другой тип. Однако такой тип можно

```
void mumble 0 {
    // правильно: программа по-прежнему содержит ошибку,
    // но теперь она компилируется!
    // Прежде всего нужно проверить
    // явные преобразования типов...

    pc = static_cast< char* >( pv );

    char *pstr = new char[ str1en( pc )+1 ];

    // скорее всего приведет к краху
    strcpy( pstr, pc );
```

изменить явно:

```
}
```

Другой причиной использования явного преобразования типов может служить необходимость избежать стандартного преобразования или выполнить вместо него собственное. Например, в следующем выражении ival сначала преобразуется в double,

```
double dval;
int iva1;
```

потом к нему прибавляется dval, и затем результат снова трансформируется в int.

```
ival += dval;
```

Можно уйти от ненужного преобразования, явно заменив dval на int:

```
ival += static_cast< int >( dval );
```

Третьей причиной является желание избежать неоднозначных ситуаций, в которых возможно несколько вариантов применения правил преобразования по умолчанию. (Мы рассмотрим этот случай в главе 9, когда будем говорить о перегруженных функциях.)

Синтаксис операции явного преобразования типов таков:

```
cast-name< type >( expression );
```

Здесь cast-name – одно из ключевых слов static_cast, const_cast, dynamic_cast или reinterpret_cast, а type – тип, к которому приводится выражение expression.

Четыре вида явного преобразования введены для того, чтобы учесть все возможные формы приведения типов. Так const_cast служит для трансформации константного типа в неконстантный и подвижного (volatile) – в неподвижный. Например:

```
extern char *string_copy( char* );
const char *pc_str;

char *pc = string_copy( const_cast< char* >( pc_str ));
```

Любое иное использование const_cast вызывает ошибку компиляции, как и попытка подобного приведения с помощью любого из трех других операторов.

С применением static_cast осуществляются те преобразования, которые могут быть

```
double d = 97.0;
```

сделаны неявно, на основе правил по умолчанию:

```
char ch = static_cast< char >( d );
```

Зачем использовать static_cast? Дело в том, что без него компилятор выдаст предупреждение о возможной потере точности. Применение оператора static_cast говорит и компилятору, и человеку, читающему программу, что программист знает об этом.

Кроме того, с помощью static_cast указатель void* можно преобразовать в указатель определенного типа, арифметическое значение – в значение перечисления (enum), а базовый класс – в производный. (О преобразованиях типов базовых и производных классов говорится в главе 19.)

Эти изменения потенциально опасны, поскольку их правильность зависит от того, какое конкретное значение имеет преобразуемое выражение в данный момент выполнения

```
enum mumble { first = 1, second, third };

extern int ival;
```

программы:

```
mumble mums_the_word = static_cast< mumble >( ival );
```

Трансформация ival в mumble будет правильной только в том случае, если ival равен 1, 2 или 3.

reinterpret_cast работает с внутренними представлениями объектов (re-interpret – другая интерпретация того же внутреннего представления), причем правильность этой

```
complex<double> *pcom;
```

операции целиком зависит от программиста. Например:

```
char *pc = reinterpret_cast< char* >( pcom );
```

Программист не должен забыть или упустить из виду, какой объект реально адресуется указателем char* pc. Формально это указатель на строку встроенного типа, и компилятор не будет препятствовать использованию pc для инициализации строки:

```
string str( pc );
```

хотя скорее всего такая команда вызовет крах программы.

Это хороший пример, показывающий, насколько опасны бывают явные преобразования типов. Мы можем присваивать указателям одного типа значения указателей совсем другого типа, и это будет работать до тех пор, пока мы держим ситуацию под контролем. Однако, забыв о подразумеваемых деталях, легко допустить ошибку, о которой компилятор не сможет нас предупредить.

Особенно трудно найти подобную ошибку, если явное преобразование типа делается в одном файле, а используется измененное значение в другом.

В некотором смысле это отражает фундаментальный парадокс языка C++: строгая проверка типов призвана не допустить подобных ошибок, в то же время наличие операторов явного преобразования позволяет "обмануть" компилятор и использовать объекты разных типов на свой страх и риск. В нашем примере мы "отключили" проверку типов при инициализации указателя `pc` и присвоили ему адрес комплексного числа. При инициализации строки `str` такая проверка производится снова, но компилятор считает, что `pc` указывает на строку, хотя, на самом-то деле, это не так!

Четыре оператора явного преобразования типов были введены в стандарт C++ как наименьшее зло при невозможности полностью запретить такое приведение. Устаревшая, но до сих пор поддерживаемая стандартом C++ форма явного преобразования выглядит так:

```
char *pc = (char*) pcom;
```

Эта запись эквивалентна применению оператора `reinterpret_cast`, однако выглядит не так заметно. Использование операторов `xxx_cast` позволяет четко указать те места в программе, где содержатся потенциально опасные трансформации типов.

Если поведение программы становится ошибочным и непонятным, возможно, в этом виноваты явные видоизменения типов указателей. Использование операторов явного преобразования помогает легко обнаружить места в программе, где такие операции выполняются. (Другой причиной непредсказуемого поведения программы может стать нечаянное уничтожение объекта (`delete`), в то время как он еще должен использоваться в работе. Мы поговорим об этом в разделе 8.4, когда будем обсуждать динамическое выделение памяти.)

Оператор `dynamic_cast` применяется при идентификации типа во время выполнения (run-time type identification). Мы вернемся к этой проблеме лишь в разделе 19.1.

4.14.4. Устаревшая форма явного преобразования

Операторы явного преобразования типов, представленные в предыдущем разделе, появились только в стандарте C++; раньше использовалась форма, теперь считающаяся устаревшей. Хотя стандарт допускает и эту форму, мы настоятельно не рекомендуем ею пользоваться. (Только если ваш компилятор не поддерживает новый вариант.)

```
// появившийся в C++ вид
type (expr);

// вид, существовавший в C
```

Устаревшая форма явного преобразования имеет два вида:

```
(type) expr;
```

и может применяться вместо операторов `static_cast`, `const_cast` и `reinterpret_cast`.

```
const char *pc = (const char*) pcom;
int ival = (int) 3.14159;
extern char *rewrite_str( char* );
char *pc2 = rewrite_str( (char*) pc );
```

Вот несколько примеров такого использования:

```
int addr_va1ue = int( &iva1 );
```

Эта форма сохранена в стандарте C++ только для обеспечения обратной совместимости с программами, написанными для C и предыдущих версий C++.

Упражнение 4.21

```
char cval; int ival;
float fval; double dva1;
```

Даны определения переменных:

```
unsigned int ui;
```

```
(a) cva1 = 'a' + 3;
(b) fval = ui - ival *  1.0;
(c) dva1 = ui * fval;
```

Какие неявные преобразования типов будут выполнены?

```
(d) cva1 = ival + fvat + dva1;
```

Упражнение 4.22

```
void *pv;          int ival;
char *pc;          double dval;
```

Даны определения переменных:

```
const string *ps;
```

```
(a)       pv = (void*)ps;
(b)       ival = int( *pc );
(c)       pv = &dva1;
```

Перепишите следующие выражения, используя операторы явного преобразования типов:

```
(d)       pc = (char*) pv;
```

4.15. Пример: реализация класса Stack

Описывая операции инкремента и декремента, для иллюстрации применения их префиксной и постфиксной формы мы ввели понятие стека. Данная глава завершается примером реализации класса `iStack` – стека, позволяющего хранить элементы типа `int`.

Как уже было сказано, с этой структурой возможны две основные операции – поместить элемент (push) и извлечь (pop) его. Другие операции позволяют получить информацию о текущем состоянии стека – пуст он (`empty()`) или полон (`full()`), сколько элементов в нем содержится (`size()`). Для начала наш стек будет предназначен лишь для элементов

```
#include <vector>

class iStack {
public:
    iStack( int capacity )
        : _stack( capacity ), _top( 0 ) {}

    bool pop( int &va1ue );
    boot push( int value );

    bool full();
    bool empty();
    void display();

    int size();

private:
    int _top;
    vector< int > _stack;
```

типа `int`. Вот объявление нашего класса:

```
};
```

В данном случае мы используем вектор фиксированного размера: для иллюстрации использования префиксных и постфиксных операций инкремента и декремента этого достаточно. (В главе 6 мы модифицируем наш стек, придав ему возможность динамически меняться.)

Элементы стека хранятся в векторе `_stack`. Переменная `_top` содержит индекс первой свободной ячейки стека. Этот индекс одновременно представляет количество заполненных ячеек. Отсюда реализация функции `size()`: она должна просто возвращать текущее значение `_top`.

```
inline int iStack::size() { return _top; };
```

`empty()` возвращает `true`, если `_top` равняется `0`; `full()` возвращает `true`, если `_top` равен `_stack.size()-1` (напомним, что индексация вектора начинается с `0`, поэтому мы

```
inline bool iStack::empty() { return _top ? false : true; }
inline bool iStack::full() {
    return _top < _stack.size()-l ? false : true;
```

должны вычесть `1`).

```
}
```

Вот реализация функций pop() и push(). Мы добавили операторы вывода в каждую из

```
bool iStack::pop( int &top_va1ue ) {
    if ( empty() )
        return false;

    top_value = _stack[ --_top ];
    cout << "iStack::pop(): " << top_value << endl;

    return true;
}

bool iStack::push( int value  ) {
    cout << "iStack::push( "  << value << " )\n";

    if ( full() )
        return false;

    _stack[ _top++ ] = value;
    return true;
```

них, чтобы следить за ходом выполнения:

```
}
```

Прежде чем протестировать наш стек на примере, добавим функцию display(), которая позволит напечатать его содержимое. Для пустого стека она выведет:

```
( 0 )
```

Для стека из четырех элементов – 0, 1, 2 и 3 – результатом функции display() будет:

```
( 4 )( bot: 0 1 2 3 :top )
```

```
void iStack::display() {
    cout << "( " << size() << " )( bot: ";

    for ( int ix = 0; ix < _top; ++ix )
              cout << _stack[ ix ] <<" ";

    cout << " :top )\n";
```

Вот реализация функции display():

```
}
```

А вот небольшая программа для проверки нашего стека. Цикл for выполняется 50 раз. Четное значение (2, 4, 6, 8 и т.д.) помещается в стек. На каждой итерации, кратной 5 (5, 10, 15...), распечатывается текущее содержимое стека. На итерациях, кратных 10 (10, 20, 30...), из стека извлекаются два элемента и его содержимое распечатывается еще раз.

```
#inc1ude <iostream>
#inc1ude "iStack.h"

int main() {
    iStack stack( 32 ) ;

    stack.display();
    for ( int ix = 1; ix < 51; ++ix )
    {
        if ( ix%2 == 0 )
            stack.push( ix );

        if ( ix%5 == 0 )
            stack.display();

        if ( ix%10 == 0 ) {
            int dummy;
            stack.pop( dummy ); stack.pop( dummy );
            stack.display();
    }

}
```

Вот результат работы программы:

```
( 0 )( bot: :top )
iStack push( 2 )
iStack push( 4 )
( 2 )( bot: 2 4 :top )
iStack push( 6 )
iStack push( 8 )
iStack push ( 10 )
( 5 )( bot: 2 4 6 8 10 :top )
iStack pop(): 10
iStack pop(): 8
( 3 )( bot: 2 4 6 :top )
iStack push( 12 )
iStack push( 14 )
( 5 )( bot: 2 4 6 12 14 :top )
iStack::push( 16 )
iStack::push( 18 )
iStack::push( 20 )
( 8 )( bot: 2 4 6 12 14 16 18 20 :top )
iStack::pop(): 20
iStack::pop(): 18
( 6 )( bot: 2 4 6 12 14 16 :top )
iStack::push( 22 )
iStack::push( 24 )
( 8 )( bot: 2 4 6 12 14 16 22 24 :top )
iStack::push( 26 )
iStack::push( 28 )
iStack::push( 30 )
( 11 )( bot: 2 4 6 12 14 16 22 24 26 28 30 :top )
iStack::pop(): 30
iStack::pop(): 28
( 9 )( bot: 2 4 6 12 14 16 22 24 26 :top )
iStack::push( 32 )
iStack::push( 34 )
( 11 )( bot: 2 4 6 12 14 16 22 24 26 32 34 :top )
iStack::push( 36 )
iStack::push( 38 )
iStack::push( 40 )
( 14 )( bot: 2 4 6 12 14 16 22 24 26 32 34 36 38 40 :top )
iStack::pop(): 40
iStack::popQ: 38
( 12 )( bot: 2 4 6 12 14 16 22 24 26 32 34 36 :top )
iStack::push( 42 )
iStack::push( 44 )
( 14 )( bot: 2 4 6 12 14 16 22 24 26 32 34 36 42 44 :top )
iStack::push( 46 )
iStack::push( 48 )
```

```
iStack::push( 50 )
( 17 )( bot: 2 4 6 12 14 16 22 24 26 32 34 36 42 44 46 48 50 :top )
iStack::pop(): 50
iStack::pop(): 48
( 15 )( bot: 2 4 6 12 14 16 22 24 26 32 34 36 42 44 46 :top )
```

Упражнение 4.23

Иногда требуется операция `peek()`, которая возвращает значение элемента на вершине стека без извлечения самого элемента. Реализуйте функцию `peek()` и добавьте к программе `main()` проверку работоспособности этой функции.

Упражнение 4.24

В чем вы видите два основных недостатка реализации класса `iStack`? Как их можно исправить?

5. Инструкции

Мельчайшей независимой частью C++ программы является инструкция. Она соответствует предложению естественного языка, но завершается точкой с запятой (;), а не точкой. Выражение C++ (например, `ival + 5`) становится простой инструкцией, если после него поставить точку с запятой. Составная инструкция – это последовательность простых, заключенная в фигурные скобки. По умолчанию инструкции выполняются в порядке записи. Как правило, последовательного выполнения недостаточно для решения реальных задач. Специальные управляющие конструкции позволяют менять порядок действий в зависимости от некоторых условий и повторять составную инструкцию определенное количество раз. Инструкции `if`, `if-else` и `switch` обеспечивают условное выполнение. Повторение обеспечивается инструкциями цикла `while`, `do-while` и `for`.

5.1. Простые и составные инструкции

Простейшей формой является пустая инструкция. Вот как она выглядит:

```
; // пустая инструкция
```

Пустая инструкция используется там, где синтаксис C++ требует употребления инструкции, а логика программы – нет. Например, в следующем цикле `while`, копирующем одну строку в другую, все необходимые действия производятся внутри круглых скобок (*условной* части инструкции). Однако согласно правилам синтаксиса C++ после `while` должна идти инструкция. Поскольку нам нечего поместить сюда (вся работа

```
while ( *string++ = inBuf++ )
```

уже выполнена), приходится оставить это место пустым:

```
    ; // пустая инструкция
```

Случайное появление лишней пустой инструкции не вызывает ошибки компиляции. Например, такая строка

```
ival = dval + sval;; // правильно: лишняя пустая инструкция
```

состоит из двух инструкций – сложения двух величин с присваиванием результата переменной `ival` и пустой.

Простая инструкция состоит из выражения, за которым следует точка с запятой.

```
// простые инструкции
int ival = 1024; // инструкция определения переменной
ival;            // выражение
ival + 5;        // еще одно выражение
```

Например:

```
ival = ival +5;  // присваивание
```

Условные инструкции и инструкции цикла синтаксически требуют употребления единственной инструкции, связанной с ними. Однако, как правило, этого недостаточно. В таких случаях употребляются *составные инструкции* – последовательность простых,

```
if ( ival0 > ival1 ) {
    // составная инструкция, состоящая
    // из объявления и двух присваиваний

    int temp = ival0;
    ival0 = ival1;
    ival1 = temp;
```

заключенная в фигурные скобки:

```
}
```

Составная инструкция может употребляться там же, где простая, и не нуждается в завершающей точке с запятой.

Пустая составная инструкция эквивалентна пустой простой. Приведенный выше пример

```
while ( *string++ = *inBuf++ )
```

с пустой инструкцией можно переписать так:

```
    {} // пустая инструкция
```

Составную инструкцию, содержащую определения переменных, часто называют блоком. Блок задает локальную область видимости в программе – идентификаторы, объявленные внутри блока (как temp в предыдущем примере), видны только в нем. (Блоки, области видимости и время жизни объектов рассматриваются в главе 8.)

5.2. Инструкции объявления

В C++ определение объекта, например

```
int ival;
```

рассматривается как инструкция *объявления* (хотя в данном случае более правильно было бы сказать *определения*). Ее можно использовать в любом месте программы, где разрешено употреблять инструкции. В следующем примере объявления помечены комментарием //#n, где n – порядковый номер.

```
#include <fstream>
#include <string>
#include <vector>
int main()
{
    string fileName; // #1

    cout << "Введите имя файла: ";
    cin >> fileName;

    if ( fileName.empty() ) {
        // странный случай
        cerr << "Пустое имя файла. Завершение работы.\n";
        return -1;
    }

    ifstream inFile( fileName.c_str() ); // #2
        if ( ! inFile ) {
        cerr << "Невозможно открыть файл.\n";
        return -2;
    }

    string inBuf;          // #3
    vector< string > text; // #4
    while ( inFile >> inBuf ) {
        for ( int ix = 0; ix < inBuf .size(); ++ix ) // #5
            // можно обойтись без ch,
            // но мы использовали его для иллюстрации
            if (( char ch = inBuf[ix] )=='.'){ // #6
                 ch = '_';
                 inBuf[ix] = ch;
            }
        text.push_back( inBuf );
    }

    if ( text.empty() )
        return 0;
    // одна инструкция объявления,
    // определяющая сразу два объекта
    vector<string>::iterator iter = text.begin(), // #7
                             iend = text.end();

    while ( iter != -iend ) {
        cout << *iter << '\n';
        ++iter;
    }
    return 0;

}
```

Программа содержит семь инструкций объявления и восемь определений объектов. Объявления действуют *локально;* переменная объявляется непосредственно перед первым использованием объекта.

В 70-е годы философия программирования уделяла особое внимание тому, чтобы определения всех объектов находились в начале программы или блока, перед исполняемыми инструкциями. (В C, например, определение переменной не является инструкцией и обязано располагаться в начале блока.) В некотором смысле это была реакция на идиому использования переменных без предварительного объявления, чреватую ошибками. Такую идиому поддерживал, например, FORTRAN.

Поскольку в C++ объявление является обычной инструкцией, ему разрешено появляться в любом месте программы, где допустимо употребление инструкции, что дает возможность использовать локальные объявления.

Необходимо ли это? Для встроенных типов данных применение локальных объявлений является скорее вопросом вкуса. Язык их поощряет , разрешая объявлять переменные внутри условных частей инструкций `if`, `if-else`, `switch`, `while`, `for`. Те программисты, которые любят этот стиль, верят, что таким образом делают свои программы более понятными.

Локальные объявления становятся необходимостью, когда мы используем объекты классов, имеющие конструкторы и деструкторы. Если мы помещаем все объявления в начало блока или функции, происходят две неприятные вещи:

- конструкторы всех объектов вызываются перед исполнением первой инструкции блока. Применение локальных объявлений позволяет “размазать” расходы на инициализацию по всему блоку;
- что более важно, блок или функция могут завершиться до того, как будут действительно использованы все объявленные в начале объекты. Скажем, наша программа из предыдущего примера имеет два аварийных выхода: при вводе пользователем пустого имени файла и при невозможности открыть файл с заданным именем. При этом последующие инструкции функции уже не выполняются. Если бы объекты `inBuf` и `next` были объявлены в начале блока, конструкторы и деструкторы этих объектов в случае ненормального завершения функции вызывались бы совершенно напрасно.

Инструкция объявления может состоять из одного или более определений. Например, в

```
// одна инструкция объявления,
// определяющая сразу два объекта
vector<string>::iterator iter = text.begin(),
```

нашей программе мы определяем два итератора вектора в одной инструкции:

```
                         lend = text.end();
```

```
vector<string>::iterator iter = text.begin();
```

Эквивалентная пара, определяющая по одному объекту, выглядит так:

```
vector<string>::iterator lend = text.end();
```

Хотя определение одного или нескольких объектов в одном предложении является скорее вопросом вкуса, в некоторых случаях – например, при одновременном определении объектов, указателей и ссылок – это может спровоцировать появление ошибок. Скажем, в следующей инструкции не совсем ясно, действительно ли программист хотел определить указатель и объект или просто забыл поставить звездочку перед вторым

```
// то ли хотел определить программист?
```

идентификатором (используемые имена переменных наводят на второе предположение):

```
string *ptrl, ptr2;
```

```
string *ptr1;
```

Эквивалентная пара инструкций не позволит допустить такую ошибку:

```
string *ptr2;
```

В наших примерах мы обычно группируем определения объектов в инструкции по

```
int aCnt=0, eCnt=0, iCnt=0, oCnt=0, uCnt=0;
```

сходству употребления. Например, в следующей паре

```
int charCnt=0, wordCnt=0;
```

первая инструкция объявляет пять очень похожих по назначению объектов – счетчиков пяти гласных латинского алфавита. Счетчики для подсчета символов и слов определяются во второй инструкции. Хотя такой подход нам кажется естественным и удобным, нет никаких причин считать его хоть чем-то лучше других.

Упражнение 5.1

Представьте себе, что вы являетесь руководителем программного проекта и хотите, чтобы применение инструкций объявления было унифицировано. Сформулируйте правила использования объявлений объектов для вашего проекта.

Упражнение 5.2

Представьте себе, что вы только что присоединились к проекту из предыдущего упражнения. Вы совершенно не согласны не только с конкретными правилами использования инструкций объявления, но и вообще с навязыванием каких-либо правил для этого. Объясните свою позицию.

5.3. Инструкция if

Инструкция `if` обеспечивает выполнение или пропуск инструкции или блока в

```
if ( условие )
```

зависимости от условия. Ее синтаксис таков:

```
    инструкция
```

условие заключается в круглые скобки. Оно может быть выражением, как в этом примере:

```
if(a+b>c) { ... }
```

или инструкцией объявления с инициализацией:

```
if ( int ival = compute_value() ){...}
```

Область видимости объекта, объявленного в условной части, ограничивается ассоциированной с `if` инструкцией или блоком. Например, такой код вызывает ошибку

```
if ( int ival = compute_value() ) {
    // область видимости ival
    // ограничена этим блоком
}

// ошибка: ival невидим
```

компиляции:

```
if ( ! ival ) ...
```

Попробуем для иллюстрации применения инструкции `if` реализовать функцию `min()`, возвращающую наименьший элемент вектора. Заодно наша функция будет подсчитывать число элементов, равных минимуму. Для каждого элемента вектора мы должны проделать следующее:

1. Сравнить элемент с текущим значением минимума.
2. Если элемент меньше, присвоить текущему минимуму значение элемента и сбросить счетчик в 1.
3. Если элемент равен текущему минимуму, увеличить счетчик на 1.
4. В противном случае ничего не делать.
5. После проверки последнего элемента вернуть значение минимума и счетчика.

```
if ( minVal > ivec[ i ] )...// новое значение minVal
```

Необходимо использовать две инструкции `if`:

```
if ( minVal == ivec[ i ] )...// одинаковые значения
```

Довольно часто программист забывает использовать фигурные скобки, если нужно

```
if ( minVal > ivec[ i ] )
    minVal = ivec[ i ];
```

выполнить несколько инструкций в зависимости от условия:

```
    occurs = 1; // не относится к if!
```

Такую ошибку трудно увидеть, поскольку отступы в записи подразумевают, что и `minVal=ivec[i]`, и `occurs=1` входят в одну инструкцию `if`. На самом же деле инструкция

```
occurs = 1;
```

не является частью `if` и выполняется безусловно, всегда сбрасывая `occurs` в 1. Вот как должна быть составлена правильная `if`-инструкция (точное положение открывающей фигурной скобки является поводом для бесконечных споров):

```
if ( minVal > ivec[ i ] )
{
    minVal = ivec[ i ];
    occurs = 1;
}
```

```
if ( minVal == ivec [ i ] )
```

Вторая инструкция `if` выглядит так:

```
    ++occurs;
```

Заметим, что порядок следования инструкций в этом примере крайне важен. Если мы будем сравнивать `minVal` именно в такой последовательности, наша функция всегда

```
if ( minVal > ivec[ i ] ) {
    minVal = ivec[ i ];
    occurs = 1;
}
//        если minVal только что получила новое значение,
//        то occurs будет на единицу больше, чем нужно
if        ( minVal == ivec[ i ] )
```

будет ошибаться на 1:

```
    ++occurs;
```

Выполнение второго сравнения не обязательно: один и тот же элемент не может одновременно быть и меньше и равен `minVal`. Поэтому появляется необходимость выбора одного из двух блоков в зависимости от условия, что реализуется инструкцией

```
if ( условие )
    инструкция1
else
```

`if-else`, второй формой `if`-инструкции. Ее синтаксис выглядит таким образом:

```
    инструкция2
```

инструкция`1` выполняется, если условие истинно, иначе переходим к *инструкция*`2`.

```
if ( minVal == ivec[ i ] )
    ++occurs;
else
if ( minVal > ivec[ i ] ) {
    minVal = ivec[ i ];
    occurs = 1;
```

Например:

```
}
```

Здесь инструкция2 сама является if-инструкцией. Если minVal меньше ivec[i], никаких действий не производится.

```
if ( minVal < ivec[ i ] )
    {} // пустая инструкция
else
if ( minVal > ivec[ i ] ) {
    minVal = ivec[ i ];
    occurs = 1;
}
else // minVal == ivec[ i ]
```

В следующем примере выполняется одна из трех инструкций:

```
    ++occurs;
```

Составные инструкции if-else могут служить источником неоднозначного толкования, если частей else больше, чем частей if. К какому из if отнести данную часть else?

```
if ( minVal <= ivec[ i ] )
    if ( minVal == ivec[ i ] )
        ++occurs;
else {
    minVal= ivec[ i ];
    occurs= 1;
```

(Эту проблему иногда называют проблемой висячего else). Например:

```
}
```

Судя по отступам, программист предполагает, что else относится к самому первому, внешнему if. Однако в C++ неоднозначность висячих else разрешается соотнесением их с последним встретившимся if. Таким образом, в действительности предыдущий

```
if ( minVal <= ivec[ i ] ) {
    if ( minVal == ivec[ i ] )
        ++occurs;
    else {
        minVal   = ivec[ i ];
        occurs   = 1;
    }
```

фрагмент означает следующее:

```
}
```

Одним из способов разрешения данной проблемы является заключение внутреннего if в фигурные скобки:

```
if ( minVal <= ivec[ i ] ) {
    if ( minVal == ivec[ i ] )
        ++occurs;
}
else {
    minVal= ivec[ i ];
    occurs= 1;
}
```

В некоторых стилях программирования рекомендуется всегда употреблять фигурные скобки при использовании инструкций `if-else`, чтобы не допустить возможности неправильной интерпретации кода.

Вот первый вариант функции `min()`. Второй аргумент функции будет возвращать количество вхождений минимального значения в вектор. Для перебора элементов массива используется цикл `for`. Но мы допустили ошибку в логике программы. Сможете

```
#include <vector>

int min( const vector<int> &ivec, int &occurs )
{
    int minVal = 0;
    occurs = 0;

    int size = ivec.size();

    for ( int ix = 0; ix < size; ++ix ) {
        if ( minVal == ivec[ ix ] )
            ++occurs;
        else
        if ( minVal > ivec[ ix ] ) {
            minVal = ivec[ ix ];
            occurs = 1;
        }
    }
    return minVal;
}
```

ли вы заметить ее?

Обычно функция возвращает только одно значение. Однако согласно нашей спецификации в точке вызова должно быть известно не только само минимальное значение, но и количество его вхождений в вектор. Для возврата второго значения мы использовали параметр типа ссылка. (Параметры-ссылки рассматриваются в разделе 7.3.) Любое присваивание значения ссылке `occurs` изменяет значение переменной, на которую она ссылается:

```
int main()
{
    int occur_cnt = 0;
    vector< int > ivec;

    // occur_cnt получает значение occurs
    // из функции min()
    int minval = min( ivec, occur_cnt );

    // ...

}
```

Альтернативой использованию параметра-ссылки является применение объекта класса pair, представленного в разделе 3.14. Функция min() могла бы возвращать два значения

```
// альтернативная реализация
// с помощью пары

#include <uti1ity>
#include <vector>

typedef pair<int,int> min_va1_pair;

min_va1_pair
min( const vector<int> &ivec )
{
    int minVal = 0;
    int occurs = 0;

    // то же самое ...

    return make_pair( minVal, occurs );
}
```

в одной паре:

К сожалению, и эта реализация содержит ошибку. Где же она? Правильно: мы инициализировали minVal нулем, поэтому, если минимальный элемент вектора больше нуля, наша реализация вернет нулевое значение минимума и нулевое значение количества вхождений.

Программу можно изменить, инициализировав minVal первым элементом вектора:

```
int minVal = ivec[0];
```

Теперь функция работает правильно. Однако в ней выполняются некоторые лишние действия, снижающие ее эффективность.

```
// исправленная версия min()
// оставляющая возможность для оптимизации ...

int minVal = ivec[0];
occurs = 0;
int       size = ivec.size();

for       ( int ix = 0; ix < size; ++ix )
{
    if ( minVal == ivec[ ix ] )
        ++occurs;

    // ...
```

Поскольку ix инициализируется нулем, на первой итерации цикла значение первого элемента сравнивается с самим собой. Можно инициализировать ix единицей и избежать ненужного выполнения первой итерации. Однако при оптимизации кода мы допустили другую ошибку (наверное, стоило все оставить как было!). Сможете ли вы ее

```
// оптимизированная версия min(),
// к сожалению, содержащая ошибку...

int minVal = ivec[0];
occurs = 0;
int       size = ivec.size();

for       ( int ix = 1; ix < size; ++ix )
{
    if ( minVal == ivec[ ix ] )
        ++occurs;

    // ...
```

обнаружить?

Если ivec[0] окажется минимальным элементом, переменная occurs не получит

```
int minVal = ivec[0];
```

значения 1. Конечно, исправить это очень просто, но сначала надо найти ошибку:

```
occurs = 1;
```

К сожалению, подобного рода недосмотры встречаются не так уж редко: программисты тоже люди и могут ошибаться. Важно понимать, что это неизбежно, и быть готовым тщательно тестировать и анализировать свои программы.

Вот окончательная версия функции min() и программа main(), проверяющая ее работу:

```
#include <iostream>
#include <vector>

int min( const vector< int > &ivec, int &occurs )
{
    int minVal = ivec[ 0 ];
    occurs = 1;

    int size = ivec.size();
    for ( int ix = 1; ix < size; ++ix )
    {
        if ( minVal == ivec[ ix ] )
            ++occurs;
        else
        if ( minVal > ivec[ ix ] ){
            minVal = ivec[ ix ];
            occurs = 1;
        }
    }
    return minVal;
}

int main()
{
    int ia[] = { 9,1,7,1,4,8,1,3,7,2,6,1,5,1 };
    vector<int> ivec( ia, ia+14 );

    int occurs = 0;
    int minVal = min( ivec, occurs );

    cout << "Минимальное значение: " << minVal
         << " встречается: " << occurs << " раз.\n";

    return 0;
}
```

Результат работы программы:

```
Минимальное значение: 1 встречается: 5 раз.
```

В некоторых случаях вместо инструкции `if-else` можно использовать более краткое и

```
template <class valueType>
inline const valueType&
min( valueType &vall, valueType &va12 )
{
    if ( vall < va12 )
        return vall;
    return va12;
}
```

выразительное условное выражение. Например, следующую реализацию функции `min()`:

можно переписать так:

```
template <class valueType>
inline const valueType&
min( valueType &vall, valueType &va12 )
{
    return ( vall < va12 ) ?  vall : va12;
}
```

Длинные цепочки инструкций `if-else`, подобные приведенной ниже, трудны для

```
if ( ch == 'a' ||
     ch == 'A' )
        ++aCnt;
else
if ( ch == 'e' ||
     ch == 'E' )
        ++eCnt;
else
if ( ch == 'i' ||
     ch == 'I' )
        ++iCnt;
else
if ( ch == 'o' ||
     ch == '0' )
        ++oCnt;
else
if ( ch == 'u' ||
     ch == 'U' )
        ++uCnt;
```

восприятия и, таким образом, являются потенциальным источником ошибок.

В качестве альтернативы таким цепочкам C++ предоставляет инструкцию `switch`. Это тема следующего раздела.

Упражнение 5.3

Исправьте ошибки в примерах:

```
(a) if ( ivall != iva12 )
        ivall = iva12
    else
        ivall = iva12 = 0;

(b) if ( ivat < minval )
        minvat = ival;
        occurs = 1;

(c) if ( int ival = get_value())
        cout << "ival = "
             << ival << endl;

    if ( ! ival )
        cout << "ival = 0\n";

(d) if ( ival = 0 )
        ival = get_value();

(e) if ( iva1 == 0 )
    else ival = 0;
```

Упражнение 5.4

Преобразуйте тип параметра `occurs` функции `min()`, сделав его не ссылкой, а простым объектом. Запустите программу. Как изменилось ее поведение?

5.4. Инструкция switch

Длинные цепочки инструкций `if-else`, наподобие приведенной в конце предыдущего раздела, трудны для восприятия и потому являются потенциальным источником ошибок. Модифицируя такой код, легко сопоставить, например, разные `else` и `if`. Альтернативный метод выбора одного их взаимоисключающих условий предлагает инструкция `switch`.

Для иллюстрации инструкции `switch` рассмотрим следующую задачу. Нам надо подсчитать, сколько раз встречается каждая из гласных букв в указанном отрывке текста. (Общеизвестно, что буква *e* – наиболее часто встречающаяся гласная в английском языке.) Вот алгоритм программы:

1. Считывать по одному символу из входного потока, пока они не кончатся.
2. Сравнить каждый символ с набором гласных.
3. Если символ равен одной из гласных, прибавить 1 к ее счетчику.
4. Напечатать результат.

Написанная программа была запущена, в качестве контрольного текста использовался раздел из оригинала данной книги. Результаты подтвердили, что буква *e* действительно самая частая:

```
aCnt: 394
eCnt: 721
iCnt: 461
oCnt: 349
uCnt: 186
```

Инструкция switch состоит из следующих частей:

- ключевого слова switch, за которым в круглых скобках идет выражение,

```
char ch;
while ( cm >> ch )
```

являющееся условием:

```
switch( ch )
```

- набора меток case, состоящих из ключевого слова case и константного выражения, с которым сравнивается условие. В данном случае каждая метка

```
case 'a':
case 'e':
case 'i':
case 'o':
```

представляет одну из гласных латинского алфавита:

```
case 'u':
```

- последовательности инструкций, соотносимых с метками case. В нашем примере с каждой меткой будет сопоставлена инструкция, увеличивающая значение соответствующего счетчика;
- необязательной метки default, которая является аналогом части else инструкции if-else. Инструкции, соответствующие этой метке, выполняются, если условие не отвечает ни одной из меток case. Например, мы можем подсчитать суммарное количество встретившихся символов, не являющихся

```
default: // любой символ, не являющийся гласной
```

гласными буквами:

```
++non_vowe1_cnt;
```

Константное выражение в метке case должно принадлежать к целому типу, поэтому

```
// неверные значения меток
case 3.14: // не целое
```

следующие строки ошибочны:

```
case ival: // не константа
```

Кроме того, две разные метки не могут иметь одинаковое значение.

Выражение условия в инструкции switch может быть сколь угодно сложным, в том числе включать вызовы функций. Результат вычисления условия сравнивается с метками case, пока не будет найдено равное значение или не выяснится, что такого значения нет. Если метка обнаружена, выполнение будет продолжено с первой инструкции после нее, если же нет, то с первой инструкции после метки default (при ее наличии) или после всей составной инструкции switch.

В отличие от if-else инструкции, следующие за найденной меткой, выполняются друг за другом, проходя все нижестоящие метки case и метку default. Об этом часто забывают. Например, данная реализация нашей программы выполняется совершенно не

```
#include <iostream>

int main()
{
    char ch;
    int aCnt=0, eCnt=0, iCnt=0, oCnt=0, uCnt=0;

    while ( cin >> ch )
        // Внимание! неверная реализация!
        switch ( ch ) {
            case 'a':
                ++aCnt;
            case 'e':
                ++eCnt;
            case 'i':
                ++iCnt;
            case 'o':
                ++oCnt;
            case 'u':
                ++uCnt;
        }

    cout << "Встретилась a: \t" << aCnt << '\n'
         << "Встретилась e: \t" << eCnt << '\n'
         << "Встретилась i: \t" << iCnt << '\n'
         << "Встретилась o: \t" << oCnt << '\n'
         << "Встретилась u: \t" << uCnt << '\n';
```

так, как хотелось бы:

```
}
```

Если значение ch равно i, выполнение начинается с инструкции после case 'i' и iCnt возрастет на 1. Однако следующие ниже инструкции, ++oCnt и ++uCnt, также выполняются, увеличивая значения и этих переменных. Если же переменная ch равна a, изменятся все пять счетчиков.

Программист должен явно дать указание компьютеру прервать последовательное выполнение инструкций в определенном месте switch, вставив break. В абсолютном большинстве случаев за каждой метке case должен следовать соответствующий break.

break прерывает выполнение switch и передает управление инструкции, следующей за закрывающей фигурной скобкой, – в данном случае производится вывод. Вот как это должно выглядеть:

```
switch ( ch ) {
    case 'a':
        ++aCnt;
        break;
    case 'e':
        ++eCnt;
        break;
    case 'i':
        ++iCnt;
        break;
    case 'o':
        ++oCnt;
        break;
    case 'u':
        ++uCnt;
        break;
}
```

Если почему-либо нужно, чтобы одна из секций не заканчивалась инструкцией `break`, то желательно написать в этом месте разумный комментарий. Программа создается не только для машин, но и для людей, и необходимо сделать ее как можно более понятной для читателя. Программист, изучающий чужой текст, не должен сомневаться, было ли нестандартное использование языка намеренным или ошибочным.

При каком условии программист может отказаться от инструкции `break` и позволить программе *провалиться* сквозь несколько меток `case`? Одним из таких случаев является необходимость выполнить одни и те же действия для двух или более меток. Это может понадобиться потому, что с `case` всегда связано только одно значение. Предположим, мы не хотим подсчитывать, сколько раз встретилась каждая гласная в отдельности, нас интересует только суммарное количество всех встретившихся гласных. Это можно

```
int vowelCnt = 0;
// ...
switch ( ch )
{
    // любой из символов a,e,1,o,u
    // увеличит значение vowelCnt
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        ++vowe1Cnt;
        break;
}
```

сделать так:

Некоторые программисты подчеркивают осознанность своих действий тем, что предпочитают в таком случае писать метки на одной строке:

```
switch ( ch )
{
    // допустимый синтаксис
    case 'a': case 'e':
    case 'i': case 'o': case 'u':
        ++vowe1Cnt;
        break;
}
```

В данной реализации все еще осталась одна проблема: как будут восприняты слова типа

```
UNIX
```

Наша программа не понимает заглавных букв, поэтому заглавные `U` и `I` не будут

```
switch ( ch ) {
    case 'a': case 'A':
        ++aCnt;
        break;
    case 'e': case 'E':
        ++eCnt;
        break;
    case 'i': case 'I':
        ++iCnt;
        break;
    case 'o': case 'O':
        ++oCnt;
        break;
    case 'u': case 'U':
        ++uCnt;
        break;
}
```

отнесены к гласным. Исправить ситуацию можно следующим образом:

Метка `default` является аналогом части `else` инструкции `if-else`. Инструкции, соответствующие `default`, выполняются, если условие не отвечает ни одной из меток `case`. Например, добавим к нашей программе подсчет суммарного количества согласных:

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

int main()
{
    char ch;
    int aCnt=0, eCnt=0, iCnt=0, oCnt=0, uCnt=0,
        consonantCount=0;

    while ( cin >> ch )
        switch ( ch ) {
            case 'a': case 'A':
                ++aCnt;
                break;
            case 'e': case 'E':
                ++eCnt;
                break;
            case 'i': case 'I':
                ++iCnt;
                break;
            case 'o': case 'O':
                ++oCnt;
                break;
            case 'u': case 'U':
                ++uCnt;
                break;
            default:
                if ( isa1pha( ch ) )
                    ++consonantCnt;
                break;
        }

    cout << "Встретилась a: \t" << aCnt << '\n'
         << "Встретилась e: \t" << eCnt << '\n'
         << "Встретилась i: \t" << iCnt << '\n'
         << "Встретилась o: \t" << oCnt << '\n'
         << "Встретилась u: \t" << uCnt << '\n'
         << "Встретилось согласных: \t" << consonantCnt
         << '\n';

}
```

`isalpha()` – функция стандартной библиотеки C; она возвращает `true`, если ее аргумент является буквой. `isalpha()` объявлена в заголовочном файле `ctype.h`. (Функции из `ctype.h` мы будем рассматривать в главе 6.)

Хотя оператор `break` функционально не нужен после последней метки в инструкции `switch`, лучше его все-таки ставить. Причина проста: если мы впоследствии захотим добавить еще одну метку после `case`, то с большой вероятностью забудем вписать недостающий `break`.

Условная часть инструкции `switch` может содержать объявление, как в следующем примере:

```
switch( int ival = get_response() )
```

`ival` инициализируется значением, получаемым от `get_response()`, и это значение сравнивается со значениями меток `case`. Переменная `ival` видна внутри блока `switch`, но не вне его.

Помещать же инструкцию объявления внутри тела блока switch не разрешается. Данный

```
case illegal_definition:
    // ошибка: объявление не может
    // употребляться в этом месте

    string file_name = get_file_name();
    // ...
```

фрагмент кода не будет пропущен компилятором:

```
    break;
```

Если бы разрешалось объявлять переменную таким образом, то ее было бы видно во всем блоке switch, однако инициализируется она только в том случае, если выполнение прошло через данную метку case.

Мы можем употребить в этом месте составную инструкцию, тогда объявление переменной file_name будет синтаксически правильным. Использование блока гарантирует, что объявленная переменная видна только внутри него, а в этом контексте

```
case ok:
{
    // ок

    string file_name = get_file_name();
    // ...
```

она заведомо инициализирована. Вот как выглядит правильный текст:

```
    break;
}
```

Упражнение 5.5

Модифицируйте программу из данного раздела так, чтобы она подсчитывала не только буквы, но и встретившиеся пробелы, символы табуляции и новой строки.

Упражнение 5.6

Модифицируйте программу из данного раздела так, чтобы она подсчитывала также количество встретившихся двухсимвольных последовательностей ff, fl и fi.

Упражнение 5.7

Найдите и исправьте ошибки в следующих примерах:

```
  switch ( ival ) {
    case 'a': aCnt++;
    case 'e': eCnt++;
    default: iouCnt++;
```

(a)

```
  }
```

(b)

```
switch ( ival ) {
  case 1:
    int ix = get_value();
    ivec[ ix ] = ival;
    break;
  default:
    ix = ivec.sizeQ-1;
    ivec[ ix ] = ival;
}
```

```
switch ( ival ) {
  case 1, 3, 5, 7, 9:
    oddcnt++;
    break;
  case 2, 4, 6, 8, 10:
    evencnt++;
    break;
}
```

(c)

```
int iva1=512 jva1=1024, kva1=4096;
int bufsize;
// ...
switch( swt ) {
  case ival:
    bufsize = ival *  sizeof( int );
    break;
  case jval:
    bufsize = jval *  sizeof( int );
    break;
  case kval:
    bufsize = kval *  sizeof( int );
    break;
}
```

(d)

```
enum { illustrator = 1, photoshop, photostyler = 2 };
switch ( ival ) {
  case illustrator:
    --i11us_1icense;
    break;
  case photoshop:
    --pshop_1icense;
    break;
  case photostyler:
    --psty1er_license;
  break;
```

(e)

```
}
```

5.5. Инструкция цикла for

Как мы видели, выполнение программы часто состоит в повторении последовательности инструкций – до тех пор, пока некоторое условие остается истинным. Например, мы читаем и обрабатываем записи файла, пока не дойдем до его конца, перебираем элементы массива, пока индекс не станет равным размерности массива минус 1, и т.д. В C++ предусмотрено три инструкции для организации циклов, в частности `for` и `while`, которые начинаются проверкой условия. Такая проверка означает, что цикл может закончиться без выполнения связанной с ним простой или составной инструкции. Третий тип цикла, `do while`, гарантирует, что тело будет выполнено как минимум один раз: условие цикла проверяется по его завершении. (В этом разделе мы детально рассмотрим цикл `for`; в разделе 5.6 разберем `while`, а в разделе 5.7 – `do while`.)

Цикл `for` обычно используется для обработки структур данных, имеющих

```
#include <vector>
int main() {
    int ia[ 10 ];

    for ( int ix = 0; ix < 10; ++-ix )
        ia[ ix ] = ix;
    vector<int> ivec( ia, ia+10 );
    vector<int>::iterator iter = ivec.begin() ;

    for ( ; iter != ivec.end(); ++iter )
        *iter *= 2;
return 0;
```

фиксированную длину, таких, как массив или вектор:

```
}
```

```
for (инструкция-инициализации; условие; выражение )
```

Синтаксис цикла `for` следующий:

```
   инструкция
```

инструкция-инициализации может быть либо выражением, либо инструкцией объявления. Обычно она используется для инициализации переменной значением, которое увеличивается в ходе выполнения цикла. Если такая инициализация не нужна или выполняется где-то в другом месте, эту инструкцию можно заменить пустой (см. второй из приведенных ниже примеров). Вот примеры правильного использования

```
// index и iter определены в другом месте
for ( index =0; ...
for ( ; /* пустая инструкция */ ...
for ( iter = ivec.begin(); ...
for ( int 1o = 0,hi = max; ...
```

инструкции-инициализации:

```
for ( char *ptr = getStr(); ...
```

условие служит для управления циклом. Пока условие при вычислении дает true, инструкция продолжает выполняться. Выполняемая в цикле инструкция может быть как простой, так и составной. Если же самое первое вычисление условия дает false,

```
(... index < arraySize; ... )
(... iter != ivec.end(); ... )
(... *stl++ = *st2++; ... )
```

инструкция не выполняется ни разу. Правильные условия можно записать так:

```
(... char ch = getNextChar(); ... )
```

Выражение вычисляется после выполнения инструкции на каждой итерации цикла. Обычно его используют для модификации переменной, инициализированной в инструкции-инициализации. Если самое первое вычисление условия дает false,

```
( ...  ...; ++-index )
( ...  ...; ptr = ptr->next )
( ...  ...; ++i, --j, ++cnt )
```

выражение не выполняется ни разу. Правильные выражения выглядят таким образом:

```
( ...  ...; ) // пустое выражение
```

```
const int sz = 24;
int ia[ sz ];
vector<int> ivec( sz );

for ( int ix = 0; ix < sz; ++ix )   {
    ivec[ ix ] = ix;
    ia[ ix ]= ix;
```

Для приведенного ниже цикла for

```
}
```

порядок вычислений будет следующим:

1. инструкция-инициализации выполняется один раз перед началом цикла. В данном примере объявляется переменная ix, которая инициализируется значением 0.
2. Вычисляется условие. Если оно равно true, выполняется составная инструкция тела цикла. В нашем примере, пока ix меньше sz, значение ix присваивается элементам ivec[ix] и ia[ix]. Когда значением условия станет false, выполнение цикла прекратится. Если самое первое вычисление условия даст false, составная инструкция выполняться не будет.
3. Вычисляется выражение. Как правило, его используют для модификации переменной, фигурирующей в инструкции-инициализации и проверяемой в условии. В нашем примере ix увеличивается на 1.

Эти три шага представляют собой полную итерацию цикла for. Теперь шаги 2 и 3 будут повторяться до тех пор, пока условие не станет равным false, т.е. ix окажется равным или большим sz.

В инструкции-инициализации можно определить несколько объектов, однако все они

```
for ( int ival = 0, *pi = &ia, &ri = val;
    ival < size;
    ++iva1, ++pi, ++ri )
```

должны быть одного типа, так как инструкция объявления допускается только одна:

```
        // ...
```

Объявление объекта в условии гораздо труднее правильно использовать: такое объявление должно хотя бы раз дать значение false, иначе выполнение цикла никогда

```
#include <iostream>

int main()
{
    for ( int ix = 0;
          bool done = ix == 10;
          ++ix )
               cout << "ix: " << ix << endl;
}
```

не прекратится. Вот пример, хотя и несколько надуманный:

Видимость всех объектов, определенных внутри круглых скобок инструкции for, ограничена телом цикла. Например, проверка iter после цикла вызовет ошибку

```
int main()
{
    string word;
    vector< string > text;
    // ...
    for ( vector< string >::iterator
                iter = text.begin(),
                iter_end = text.end();
          iter != text.end(); ++iter )
    {
        if ( *iter == word )
            break;
        // ...
    }

    // ошибка: iter и iter_end невидимы
    if ( iter != iter_end )
```

компиляции[8]:

Примечание [O.A.2]: Нумерация сносок сбита, как и вся остальная. Необходима проверка.

8 До принятия стандарта языка C++ видимость объектов, определенных внутри круглых скобок for, простиралась на весь блок или функцию, содержащую данную инструкцию. Например, употребление двух циклов for внутри одного блока

```
        // ...
```

Упражнение 5.8

(a)

```
for ( int *ptr = &ia, ix = 0;
         ix < size && ptr != ia+size;
         ++ix, ++ptr )
            // ...
```

Допущены ли ошибки в нижеследующих циклах for? Если да, то какие?

(b)

```
for ( ; ; ) {
       if ( some_condition )
            break;
       // ...


   }
```

```
{
    // верно для стандарта C++
    // в предыдущих версиях C++ - ошибка: ival определена дважды
    for (int ival = 0; ival < size; ++iva1 ) // ...
    for (int ival = size-1; ival > 0; ival ) // ...
}
```

в ранних версиях языка вызывало ошибку: `ival` определена дважды. В стандарте C++ данный текст синтаксически правилен, так как каждый экземпляр `ival` является локальным для своего блока.

(c)

```
for ( int ix = 0; ix < sz; ++ix )
      // ...

   if ( ix != sz )
      // ...
```

(d)

```
int ix;
   for ( ix < sz; ++ix )
      // ...
```

(e)

```
for ( int ix = 0; ix < sz; ++ix, ++ sz )

      // ...
```

Упражнение 5.9

Представьте, что вам поручено придумать общий стиль использования цикла `for` в вашем проекте. Объясните и проиллюстрируйте примерами правила использования каждой из трех частей цикла.

Упражнение 5.10

```
bool is_equa1( const vector<int> &vl,
```

Дано объявление функции:

```
               const vector<int> &v2 );
```

Напишите тело функции, определяющей равенство двух векторов. Для векторов разной длины сравнивайте только то количество элементов, которое соответствует меньшему из двух. Например, векторы (0,1,1,2) и (0,1,1,2,3,5,8) считаются равными. Длину векторов можно узнать с помощью функций `v1.size()` и `v2.size()`.

5.6. Инструкция while

```
while ( условие )
```

Синтаксис инструкции `while` следующий:

```
    инструкция
```

Пока значением *условия* является `true`, *инструкция* выполняется в такой последовательности:

1. Вычислить *условие*.
2. Выполнить *инструкцию*, если *условие* истинно.

3. Если самое первое вычисление условия дает false, инструкция не выполняется.

```
bool quit = false;
// ...
while ( ! quit ) {
    // ...
    quit = do_something();
}

string word;
```

Условием может быть любое выражение:

```
while ( cin >> word ){ ... }
```

```
while ( symbol *ptr = search( name )) {
    // что-то сделать
}
```

или объявление с инициализацией:

В последнем случае ptr видим только в блоке, соответствующем инструкции while, как это было и для инструкций for и switch.

Вот пример цикла while, обходящего множество элементов, адресуемых двумя

```
int sumit( int *parray_begin, int *parray_end )
{
    int sum = 0;

    if ( ! parray_begin || ! parray_end )
        return sum;

    while ( parray_begin != parray_end )
        // прибавить к sum
        // и увеличить указатель
        sum += *parray_begin++;

    return sum;
}

int       ia[6] = { 0, 1, 2, 3, 4, 5 };
int       main()
{
    int sum = sumit( &ia[0], &ia[ 6 ] );
    // ...
}
```

указателями:

Для того чтобы функция sumit() выполнялась правильно, оба указателя должны адресовать элементы *одного и того же* массива (parray_end может указывать на элемент, следующий за последним). В противном случае sumit() будет возвращать бессмысленную величину. Увы, C++ не гарантирует, что два указателя адресуют один и

тот же массив. Как мы увидим в главе 12, стандартные универсальные алгоритмы реализованы подобным же образом, они принимают параметрами указатели на первый и последний элементы массива.

Упражнение 5.11

(a)

```
string bufString, word;
while ( cin >> bufString >> word )
```

Какие ошибки допущены в следующих циклах `while`:

(b)

```
while ( vector<int>::iterator iter !=  ivec.end() )
    // ...
```

(c)

```
while ( ptr = 0 )
    ptr = find_a_value();
```

(d)

```
while ( bool status = find( word )) {
    word = get_next_word();
    if ( word.empty() )
        break;
    // ...
}
if ( ! status )

    // ...

    cout << "Слов не найдено\n";
```

Упражнение 5.12

`while` обычно применяется для циклов, выполняющихся, пока некоторое условие истинно, например, читать следующее значение, пока не будет достигнут конец файла. `for` обычно рассматривается как пошаговый цикл: индекс пробегает по определенному диапазону значений. Напишите по одному типичному примеру `for` и `while`, а затем измените их, используя цикл другого типа. Если бы вам нужно было выбрать для постоянной работы только один из этих типов, какой бы вы выбрали? Почему?

Упражнение 5.13

Напишите функцию, читающую последовательность строк из стандартного ввода до тех пор, пока одно и то же слово не встретится два раза подряд либо все слова не будут обработаны. Для чтения слов используйте `while`; при обнаружении повтора слова завершите цикл с помощью инструкции `break`. Если повторяющееся слово найдено, напечатайте его. В противном случае напечатайте сообщение о том, что слова не повторялись.

5.8. Инструкция do while

Представим, что нам надо написать программу, переводящую мили в километры.

```
int val;
bool more = true; // фиктивное значение, нужное для
                  // начала цикла

while ( more ) {
    val = getValue();
    val = convertValue(val);
    printValue(val);
    more = doMore();
```

Структура программы выглядит так:

```
}
```

Проблема заключается в том, что условие вычисляется в теле цикла. `for` и `while` требуют, чтобы значение условия равнялось `true` до первого вхождения в цикл, иначе тело не выполнится ни разу. Это означает, что мы должны обеспечить такое условие до начала работы цикла. Альтернативой может служить использование `do while`, гарантирующего выполнение тела цикла хотя бы один раз. Синтаксис цикла `do while`

```
do
    инструкция
```

таков:

```
while ( условие );
```

инструкция выполняется до первой проверки *условия*. Если вычисление *условия* дает `false`, цикл останавливается. Вот как выглядит предыдущий пример с использованием

```
do {
    val = getValue();
    val = convertValue(val);
    printValue(val);
```

цикла `do while`:

```
} while doMore();
```

В отличие от остальных инструкций циклов, `do while` не разрешает объявлять объекты в

```
// ошибка: объявление переменной
// в условии не разрешается
do {
    // ...
    mumble( foo );
```

своей части условия. Мы не можем написать:

```
} while ( int foo = get_foo() ) // ошибка
```

потому что до условной части инструкции do while мы дойдем только после первого выполнения тела цикла.

Упражнение 5.14

Какие ошибки допущены в следующих циклах do while:

```
do
    string rsp;
    int vail, va12;
    cout << "Введите два числа: ";
    c-in >> vail >> va12;
    cout << "Сумма " << vail
         << " и " << va12
         << " = " << vail + va12 << "\n\n"
         << "Продолжить? [да][нет] ";
    cin >> rsp;
while ( rsp[0] != 'n' );
```

(b)

```
do {
    // ...
} while ( int iva1 = get_response() );
```

(c)

```
 do {
        int ival = get_response();
        if ( iva1 == some_value() )
            break;
    } while ( iva1 );

    if ( !iva1 )
```

(a)

```
    // ...
```

Упражнение 5.15

Напишите небольшую программу, которая запрашивает у пользователя две строки и печатает результат лексикографического сравнения этих строк (строка считается меньшей, если идет раньше при сортировке по алфавиту). Пусть она повторяет эти действия, пока пользователь не даст команду закончить. Используйте тип string, сравнение строк и цикл do while.

5.8. Инструкция break

Инструкция break останавливает циклы for, while, do while и блока switch. Выполнение программы продолжается с инструкции, следующей за закрывающей фигурной скобкой цикла или блока. Например, данная функция ищет в массиве целых чисел определенное значение. Если это значение найдено, функция сообщает его индекс, в противном случае она возвращает -1. Вот как выглядит реализация функции:

```
// возвращается индекс элемента или  -1
int search( int *ia, int size, int value )
{
    // проверка что ia != 0 и size > 0 ...

    int loc = -1;
    for ( int ix = 0; ix < size; ++ix ) {
        if ( value == ia[ ix ] ) {
            // нашли!
            // запомним индекс и выйдем из цикла
            loc = ix;
            break;
        }
    } // конец цикла

    // сюда попадаем по break ...
    return loc;

}
```

В этом примере break прекращает выполнение цикла for и передает управление инструкции, следующей за этим циклом, – в нашем случае return. Заметим, что break выводит из блока, относящегося к инструкции for, а не if, хотя является частью составной инструкции, соответствующей if. Использование break внутри блока if, не

```
// ошибка: неверное использование break
if ( ptr ) {
    if ( *ptr == "quit" )
        break;
    // ...
}
```

входящего в цикл или в switch, является синтаксической ошибкой:

Если эта инструкция используется внутри вложенных циклов или инструкций switch, она завершает выполнение того внутреннего блока, в котором находится. Цикл или switch, включающий тот цикл или switch, из которого мы вышли с помощью break,

```
white ( cin >> inBuf )
{
    switch( inBuf[ 0 ] ) {
    case '-':
        for ( int ix = 1; ix < inBuf.size(); ++ix ) {
            if ( inBuf[ ix ] == ' ' )
                break; // #1
            // ...
        // ...
        }
        break; // #2
    case '+':
        // ...
    }
```

продолжает выполняться. Например:

Инструкция break, помеченная // #1, завершает выполнение цикла for внутри ветви case '-' блока switch, но не сам switch. Аналогично break // #2 завершает выполнение блока switch, но не цикла while, в который тот входит.

5.9. Инструкция continue

Инструкция continue завершает текущую итерацию цикла и передает управление на вычисление условия, после чего цикл может продолжиться. В отличие от инструкции break, завершающей выполнение всего цикла, инструкция continue завершает выполнение только текущей итерации. Например, следующий фрагмент программы читает из входного потока по одному слову. Если слово начинается с символа подчеркивания, оно обрабатывается, в противном случае программа переходит к новому

```
while ( cin >> inBuf ) {
    if ( inBuf[0] '= '_' )
        continue; // завершение итерации

    // обработка слова ...
```

слову.

```
}
```

Инструкция continue может быть использована только внутри цикла.

5.10. Инструкция goto

Инструкция goto обеспечивает безусловный переход к другой инструкции внутри той же функции, поэтому современная практика программирования выступает против ее применения.

Синтаксис goto следующий:

```
goto метка;
```

где метка – определенный пользователем идентификатор. Метка ставится перед инструкцией, на которую можно перейти с помощью goto, и должна заканчиваться двоеточием. Нельзя ставить метку непосредственно перед закрывающей фигурной

```
end: ; // пустая инструкция
```

скобкой. Если же это необходимо, их следует разделить пустой инструкцией:

```
}
```

Переход через инструкцию объявления в том же блоке с помощью goto невозможен. Например, данная функция вызывает ошибку компиляции:

```
int oops_in_error() {
    // mumble ...
    goto end;
    // ошибка: переход через объявление
    int ix = 10;
    // ... код, использующий ix
end: ;
}
```

Правильная реализация функции помещает объявление `ix` и использующие его

```
int oops_in_error() {
    // mumble ...
    goto end;

    {
        // правильно: объявление во вложенном блоке
        int ix = 10;
        // ... код, использующий ix
    }
end: ;
}
```

инструкции во вложенный блок:

Причина такого ограничения та же, что и для объявлений внутри блока `switch`: компилятор должен гарантировать, что для объявленного объекта конструктор и деструктор либо выполняются вместе, либо ни один из них не выполняется. Это и достигается заключением объявления во вложенный блок.

Переход назад через объявление, однако, не считается ошибкой. Почему? Перепрыгнуть через инициализацию объекта нельзя, но проинициализировать один и тот же объект несколько раз вполне допустимо, хотя это может привести к снижению эффективности. Например:

```
// переход назад через объявление не считается ошибкой.
void
mumble ( int max_size )
{
  begin:
    int sz = get_size();
    if ( sz <= 0 ) {
        // выдать предупреждение ...
        goto end;
    }
    else
    if ( sz > max_size )
        // получить новое значение sz
        goto begin;
{ // правильно: переход через целый блок
        int ia = new int[ sz ];
        doit( ia, sz ) ;
        delete [] ia;
    }
  end:
    ;
}
```

Использование инструкции `goto` резко критикуется во всех современных языках программирования. Ее применение приводит к тому, что ход выполнения программы становится трудно понять и, следовательно, такую программу трудно модифицировать. В большинстве случаев `goto` можно заменить на инструкции `if` или циклы. Если вы все-таки решили использовать `goto`, не перескакивайте через большой фрагмент кода, чтобы можно было легко найти начало и конец вашего перехода.

5.11. Пример связанного списка

Мы завершали главы 3 и 4 примерами для введения читателя в механизм классов C++. В конце этого раздела мы покажем, как разработать класс, представляющий собой односвязный список. (В главе 6 мы рассмотрим двусвязный список, являющийся частью стандартной библиотеки.) Если вы в первый раз читаете эту книгу, то можете пропустить данный раздел и вернуться к нему после чтения главы 13. (Для усвоения этого материала нужно представлять себе механизм классов C++, конструкторы, деструкторы и т.д. Если вы плохо знаете классы, но все же хотите продолжить чтение данного раздела, мы рекомендуем прочесть пункты 2.3 и 3.15.

Список представляет собой последовательность элементов, каждый из которых содержит значение некоторого типа и адрес следующего элемента (причем для последнего из них адрес может быть нулевым). К любой такой последовательности всегда можно добавить еще один элемент (хотя реальная попытка подобного добавления может закончиться неудачно, если отведенная программе свободная память исчерпана). Список, в котором нет ни одного элемента, называется пустым.

Какие операции должен поддерживать список? Добавление (insert), удаление (remove) и поиск (find) определенных элементов. Кроме того, можно запрашивать размер списка (size), распечатывать его содержимое (display), проверять равенство двух списков. Мы покажем также, как инвертировать (reverse) и сцеплять (concatenate) списки.

Простейшая реализация операции `size()` перебирает все элементы, подсчитывая их количество. Более сложная реализация сохраняет размер как член данных; она намного

эффективнее, однако требует некоторого усложнения операций `insert()` и `remove()` для поддержки размера в актуальном состоянии.

Мы выбрали второй вариант реализации функции `size()` и храним размер списка в члене данных. Мы предполагаем, что пользователи будут достаточно часто применять эту операцию, поэтому ее необходимо реализовать как можно более эффективно.

(Одним из преимуществ отделения открытого интерфейса от скрытой реализации является то, что если наше предположение окажется неверным, мы сможем переписать реализацию, сохранив открытый интерфейс – в данном случае тип возвращаемого значения и набор параметров функции `size()` – и программы, использующие эту функцию, не нужно будет модифицировать.)

Операция `insert()` в общем случае принимает два параметра: указатель на один из элементов списка и новое значение, которое вставляется после указанного элемента. Например, для списка

```
1 1 2 3 8
```

вызов

```
mylist.insert (pointer_to_3, 5);
```

изменит наш список так:

```
1 1 2 3 5 8
```

Чтобы обеспечить подобную возможность, нам необходимо дать пользователю способ получения адреса определенного элемента. Одним из способов может быть использование функции `find()` – нахождение элемента с определенным значением:

```
pointer_to_3 = mylist.find( 3 );
```

`find()` принимает в качестве параметра значение из списка. Если элемент с таким значением найден, то возвращается его адрес, иначе `find()` возвращает 0.

Может быть два специальных случая вставки элемента: в начало и в конец списка. Для

```
insert_front( value );
```

этого требуется только задание значения:

```
1nsert_end( value );
```

Предусмотрим следующие операции удаления элемента с заданным значением, первого

```
remove( value );
remove_front();
```

элемента и всех элементов списка:

```
remove_all();
```

Функция `display()` распечатывает размер списка и все его элементы. Пустой список можно представить в виде:

```
(0)( )
```

а список из семи элементов как:

```
 (7) ( 0 1 1 2 3 5 8 )
```

`reverse()` меняет порядок элементов на противоположный. После вызова

```
mylist.reverse();
```

предыдущий список выглядит таким образом:

```
 (7) ( 8 5 3 2 1 1 0 )
```

Конкатенация добавляет элементы второго списка в конец первого. Например, для двух списков:

```
(4)( 0 1 1 2 ) // listl
(4)( 2 3 5 8 ) // list2
```

операция

```
listl.concat( list2 );
```

превращает `list1` в

```
(8) ( 0 1 1 2 2 3 5 8 )
```

Чтобы сделать из этого списка последовательность чисел Фибоначчи, мы можем воспользоваться функцией `remove()`:

```
listl.remove( 2 );
```

Мы определили поведение нашего списка, теперь можно приступать к реализации. Пусть список (`list`) и элемент списка (`list_item`) будут представлены двумя разными классами. (Ограничимся теми элементами, которые способны хранить только целые значения. Отсюда названия наших классов – `ilist` и `ilist_item`.)

Наш список содержит следующие члены: `_at_front` – адрес первого элемента, `_at_end` – адрес последнего элемента и `_size` – количество элементов. При определении объекта типа `ilist` все три члена должны быть инициализированы `0`. Это обеспечивается конструктором по умолчанию:

```
class ilist_item;

class ilist {
public:
    // конструктор по умолчанию
    ilist() : _at_front( 0 ),
              _at_end( 0 ), _size( 0 ) {}

    // ...
private:
    ilist_item *_at_front;
    ilist_item *_at_end;
    int _size;

};
```

Теперь мы можем определять объекты типа `ilist`, например:

```
ilist mylist;
```

но пока ничего больше. Добавим возможность запрашивать размер списка. Включим объявление функции `size()` в открытый интерфейс списка и определим эту функцию так:

```
inline int ilist::size() { return _size; }
```

Теперь мы можем использовать:

```
int size = mylist.size();
```

Пока не будем позволять присваивать один список другому и инициализировать один список другим (впоследствии мы реализуем и это, причем такие изменения не потребуют модификации пользовательских программ). Объявим копирующий конструктор и копирующий оператор присваивания в закрытой части определения списка без их

```
class ilist {
public:
    // определения не показаны
    ilist();
    int size();
    // ...
private:
    // запрещаем инициализацию
    // и присваивание одного списка другому
ilist( const ilist& );
ilist& operator=( const ilist& );

    // данные-члены без изменения
```

реализации. Теперь определение класса `ilist` выглядит таким образом:

```
};
```

Обе строки следующей программы вызовут ошибки компиляции, потому что функция `main()` не может обращаться к закрытым членам класса `ilist`:

```
int main()
{
    ilist yourlist( mylist ); // ошибка
    mylist = mylist;          // ошибка
}
```

Следующий шаг – вставка элемента, для представления которого мы выбрали отдельный

```
class ilist_item {
public:
    // ...
private:
    int         _value;
    ilist_item *_next;
```

класс:

```
};
```

Член `_value` хранит значение, а `_next` – адрес следующего элемента или `0`.

Конструктор `ilist_item` требует задания значения и необязательного параметра – адреса существующего объекта `ilist_item`. Если этот адрес задан, то создаваемый объект `ilist_item` будет помещен в список после указанного. Например, для списка

```
0 1 1 2 5
```

вызов конструктора

```
ilist_item ( 3, pointer_to_2 );
```

модифицирует последовательность так:

```
0 1 1 2 3 5
```

Вот реализация `ilist_item`. (Напомним, что второй параметр конструктора является необязательным. Если пользователь не задал второй аргумент при вызове конструктора, по умолчанию употребляется 0. Значение по умолчанию указывается в объявлении функции, а не в ее определении; это поясняется в главе 7.)

```
class ilist_item {
public:
    ilist_item( int value, ilist_-item *item_to_link_to = 0 );
    // ...
};

inline
ilist_item::
ilist_item( int value,  ilist_item *item )
          : _value( value )
{
    if ( item )
        _next = 0;
    else {
        _next = item->_next;
        item->_next = this;
}
```

Операция `insert()` в общем случае работает с двумя параметрами – значением и адресом элемента, после которого производится вставка. Наш первый вариант

```
inline void
ilist::
insert( ilist_item *ptr, int value )
{
    new ilist_item( value, ptr );
    ++_size;
}
```

реализации имеет два недочета. Сможете ли вы их найти?

Одна из проблем заключается в том, что указатель не проверяется на нулевое значение. Мы обязаны распознать и обработать такую ситуацию, иначе это приведет к краху программы во время исполнения. Как реагировать на нулевой указатель? Можно аварийно закончить выполнение, вызвав стандартную функцию `abort()`, объявленную в

```
#include <cstdlib>
// ...
if ( ! ptr )
```

заголовочном файле `cstdlib`:

```
    abort();
```

Кроме того, можно использовать макрос `assert()`. Это также приведет к аварийному

```
#include <cassert>
// ...
```

завершению, но с выводом диагностического сообщения:

```
assert( ptr != 0 );
```

Третья возможность – возбудить исключение:

```
if ( ! ptr )
    throw "Panic: ilist::insert(): ptr == 0";
```

В общем случае желательно избегать аварийного завершения программы: в такой ситуации мы заставляем пользователя беспомощно сидеть и ждать, пока служба поддержки обнаружит и исправит ошибку.

Если мы не можем продолжать выполнение там, где обнаружена ошибка, лучшим решением будет возбуждение исключения: оно передает управление вызвавшей программе в надежде, что та сумеет выйти из положения.

Мы же поступим совсем другим способом: рассмотрим передачу нулевого указателя как

```
if ( ! ptr )
```

запрос на вставку элемента перед первым в списке:

```
    insert_front( value );
```

Второй изъян в нашей версии можно назвать философским. Мы реализовали `size()` и `_size` как пробный вариант, который может впоследствии измениться. Если мы преобразуем функции `size()` таким образом, что она будет просто пересчитывать элементы списка, член `_size` перестанет быть нужным. Написав:

```
++_size;
```

мы тесно связали реализацию `insert()` с текущей конструкцией алгоритма пересчета элементов списка. Если мы изменим алгоритм, нам придется переписывать эту функцию, как и `insert_front()`, `insert_end()` и все операции удаления из списка. Вместо того чтобы распространять детали текущей реализации на разные функции класса, лучше

```
inline void ilist::bump_up_size()   { ++_size; }
```

инкапсулировать их в паре:

```
inline void ilist::bump_down_size() { --_size; }
```

Поскольку мы объявили эти функции встроенными, эффективность не пострадала. Вот

```
inline void
ilist::
insert( ilist_item *ptr, int value )
if (      !ptr )
    insert_front( value );
else      {
    bump_up_size();
    new ilist_item( value, ptr );
    }
```

окончательный вариант `insert()`:

```
}
```

Реализация функций `insert_front()` и `insert_end()` достаточно очевидна. В каждой

```
inline void
ilist::
insert_front( int value )
{
    ilist_item *ptr = new ilist_item( value );

    if ( !_at_front )
        _at_front = _at_end = ptr;
    else {
        ptr->next( _at_front );
        _at_front = ptr;
    }
    bump_up_size();
}

inl-ine void
ilist::
insert_end( int value )
{
    if ( !_at_end )
        _at_end = _at_front = new ilist_item( value );
    else _at_end = new ilist_item(   value, _at_end );

    bump_up_s-ize();
```

из них мы должны предусмотреть случай, когда список пуст.

```
}
```

`find()` ищет значение в списке. Если элемент с указанным значением найден,

```
ilist_item*
ilist::
find( int value )
{
    ilist_item *ptr = _at_front;
    while ( ptr )
    {
        if ( ptr->value() == value )
            break;
            ptr = ptr->next();
    }
    return ptr;
```

возвращается его адрес, иначе `find()` возвращает `0`. Реализация `find()`выглядит так:

```
}
```

```
ilist_item *ptr = mylist.find( 8 );
```

Функцию `find()` можно использовать следующим образом:

```
mylist.insert( ptr, some_value );
```

или в более компактной записи:

```
mylist.insert( mylist.find( 8 ), some_value );
```

Перед тем как тестировать операции вставки элементов, нам нужно написать функцию `display()`, которая поможет нам при отладке. Алгоритм `display()` достаточно прост: печатаем все элементы, с первого до последнего. Можете ли вы сказать, где в данной

```
// не работает правильно!

for ( ilist_item *iter = _at_front; // начнем с первого
      iter != _at_end;              // пока не последний
      ++iter )                      // возьмем следующий
        cout << iter->value() << ' ';

// теперь напечатаем последний
```

реализации ошибка?

```
cout << iter->value();
```

Список – это не массив, его элементы не занимают непрерывную область памяти. Инкремент итератора

```
++iter;
```

вовсе не сдвигает его на следующий элемент списка. Вместо этого он указывает на место в памяти, непосредственно следующее за данным элементом, а там может быть все что угодно. Для изменения значения итератора нужно воспользоваться членом `_next` объекта `ilist_item`:

```
iter = iter->_next;
```

Мы инкапсулировали доступ к членам `ilist_item` набором встраиваемых функций.

```
class ilist_item {
public:
    ilist_item( int value, ilist_item *item_to_link_to = 0 );
    int         value() { return _value; }
    iilst_item* next()  { return _next; }

    void next( ilist_item *link ) { _next = link; }
    void value( int new_value ) { _value = new_value; }

private:
    int _value;
    ilist_item *_next;
```

Определение класса `ilist_item` теперь выглядит так:

```
};
```

Вот определение функции `display()`, использующее последнюю реализацию класса `ilist_item`:

```
#include <iostream>

class ilist {
public:
    void display( ostream &os = cout );
    // ...
};

void ilist::
display( ostream &os )
{
    os << "\n( " << _size << " )(    ";

    ilist_item *ptr = _at_front;
    while ( ptr ) {
        os << ptr->value() << " ";
        ptr = ptr->next();
    }

    os << ")\n";
}
```

Тестовую программу для нашего класса `ilist` в его текущей реализации можно представить таким образом:

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

int main()
{
  ilist mylist;

  for ( int ix = 0; ix < 10; ++ix ) {
    mylist.insert_front( ix );
    mylist.insert_end( ix );
  }
  cout <<
       "Ok: после insert_front() и insert_end()\n";
  mylist.display();

  ilist_item *it = mylist.find( 8 );
  cout << "\n"
       << "Ищем значение 8: нашли?"
       << ( it ? " да!\n" : " нет!\n" );

  mylist.insert( it, 1024 );
  cout << "\n" <<
  "Вставка элемента 1024 после 8\n";

  mylist.display();
  int elem_cnt = mylist.remove( 8 );
  cout << "\n"
     << "Удалено " << elem_cnt
     << " элемент(ов) со значением 8\n";

  mylist.display();
  cout << "\n" << "Удален первый элемент\n";

  mylist.remove_front(); mylist.display();

  cout << "\n" << "Удалены все элементы\n";
  mylist.remove_all(); mylist.display();
}
```

Результат работы программы:

```
Ok: после insert_front() и insert_end()

(20)( 9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 )

Ищем значение 8: нашли? да!

Вставка элемента 1024 после 8

( 21 )( 9 8 1024 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 )

Удалено 2 элемент(ов) со значением 8

( 19 )( 9 1024 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 9 )

Удален первый элемент

( 18 )( 1024 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 9 )

Удалены все элементы

( 0 )( )
```

Помимо вставки элементов, необходима возможность их удаления. Мы реализуем три

```
void remove_front();
void remove_all ();
```

таких операции:

```
int remove( int value );
```

```
inline void
i1ist::
remove_front()
{
    if ( _at_front ) {
        ilist_item *ptr = _at_front;
        _at_front = _at_front->next();

        bump_down_size() ;
        delete ptr;
    }
```

Вот как выглядит реализация `remove_front()`:

```
}
```

`remove_all()` вызывает `remove_front()` до тех пор, пока все элементы не будут

```
void ilist::
remove_all()
{
    while ( _at_front )
        remove_front();

    _size = 0;
    _at_front = _at_end = 0;
```

удалены:

```
}
```

Общая функция `remove()` также использует `remove_front()` для обработки специального случая, когда удаляемый элемент (элементы) находится в начале списка. Для удаления из середины списка используется итерация. У элемента, предшествующего удаляемому, необходимо модифицировать указатель `_next`. Вот реализация функции:

```
int ilist::
remove( int value )
{
    ilist_item *plist = _at_front;
    int elem_cnt = 0;

    while ( plist && plist->value() == value )
    {
        plist = plist->next();
        remove_front();
        ++elem_cnt;
    }

    if ( ! plist )
        return elem_cnt;

    ilist_item *prev = plist;
    plist = plist->next();

    while ( plist ) {
    if ( plist->value() == value ) {
        prev->next( plist->next() );
        delete plist;
        ++elem_cnt;
        bump_down_size();
        plist = prev->next();
        if ( ! plist ) {
            _at_end = prev;
            return elem_cnt;
        }
    }
    else {
        prev = plist;
        plist = plist->next();
    }

    return elem_cnt;
}
```

Следующая программа проверяет работу операций в четырех случаях: когда удаляемые элементы расположены в конце списка, удаляются все элементы, таких элементов нет или они находятся и в начале, и в конце списка.

```
#include <iostream>
#include "ilist.h"
int main()
{
  ilist mylist;
   cout << "\n-----------------------------------------------\n"
        << "тест #1: - элементы в конце\n"
        << "-----------------------------------------------\n";

   mylist.insert_front( 1 ); mylist.insert_front( 1 );
   mylist.insert_front( 1 );

   my1ist.insert_front( 2 ); mylist.insert_front( 3 );
   my1ist.insert_front( 4 );

   mylist.display();

   int elem_cnt = mylist.remove( 1 );
   cout << "\n" << "Удалено " << elem_cnt
        << " элемент(ов) со значением 1\n";

   mylist.display();

   mylist.remove_all();

   cout << "\n-----------------------------------------------\n"
        << "тест #2: - элементы в начале\n"
        << "-----------------------------------------------\n";

   mylist.insert_front( 1 ); mylist.insert_front( 1 );
   mylist.insert_front( 1 );

   mylist.display();

   elem_cnt = mylist.remove( 1 );

   cout << "\n" << "Удалено " << elem_cnt
        << " элемент(ов) со значением 1\n";
   mylist.display();

   mylist.remove_all () ;

   cout << "\n-----------------------------------------------\n"
        << "тест #3: - элементов нет в списке\n"
        << "-----------------------------------------------\n";

   mylist.insert_front( 0 ); mylist.insert_front( 2 );
   mylist.insert_front( 4 );

   mylist.display();

   elem_cnt = mylist.remove( 1 );

   cout << "\n" << "Удалено " << elem_cnt
        << " элемент(ов) со значением 1\n";
   mylist.display();

   mylist.remove_all () ;

   cout << "\n-----------------------------------------------\n"
        << "тест #4: - элементы в конце и в начале\n"
        << "-----------------------------------------------\n";
   my1ist.insert_front( 1 ); mylist.insert_front( 1 );
   my1ist.insert_front( 1 );

   my1ist.insert_front( 0 ); mylist.insert_front( 2 );
   my1ist.insert_front( 4 );

   mylist.insert_front( 1 ); my1ist.insert_front( 1 );
   mylist.insert_front( 1 );

   mylist.display() ;

   elem_cnt = mylist.remove( 1 );

   cout << "\n" << "Удалено " << elem_cnt
        << " элемент(ов) со значением 1\n";
```

```
}
```

Результат работы программы:

```
-----------------------------------------------
тест #1: - элементы в конце
-----------------------------------------------

( 6 )( 4 3 2 1 1 1 )

Удалено 3 элемент(ов) со значением 1

( 3 )( 4 3 2 )

-----------------------------------------------
тест #2: - элементы в начале
-----------------------------------------------

( 3 )( 1 1 1 )

Удалено 3 элемент(ов) со значением 1

( 0 )( )

-----------------------------------------------
тест #3: - элементов нет в списке
-----------------------------------------------

( 3 )( 4 2 0 )

Удалено 0 элемент(ов) со значением 1

( 3 )( 4 2 0 )

-----------------------------------------------
тест #4: - элементы в конце и в начале
-----------------------------------------------

(9 )( 1 1 1 4 2 0 1 1 1 )

Удалено 6 элемент(ов) со значением 1

( 3 )( 4 2 0 )
```

Последние две операции, которые мы хотим реализовать, – конкатенация двух списков (добавление одного списка в конец другого) и инверсия (изменение порядка элементов на противоположный). Первый вариант `concat()` содержит ошибку. Сможете ли вы ее

```
void ilist::concat( const ilist &i1 ) {
    if ( ! _at_end )
        _at_front = i1._at_front;
    else _at_end->next( i1._at_front );
    _at_end = i1._at_end;
```

найти?

```
}
```

Проблема состоит в том, что теперь два объекта `ilist` содержат последовательность одних и тех же элементов. Изменение одного из списков, например вызов операций `insert()` и `remove()`, отражается на другом, приводя его в рассогласованное состояние. Простейший способ обойти эту проблему – скопировать каждый элемент второго списка. Сделаем это при помощи функции `insert_end()`:

```
void ilist::
concat( const ilist &i1 )
{
    i1ist_item *ptr = i1._at_front;
    while ( ptr ) {
        insert_end( ptr->value() );
        ptr = ptr->next();
    }
}
```

```
void
ilist::
reverse()
{
    ilist_item *ptr = _at_front;
    ilist_item *prev = 0;

    _at_front = _at_end;
    _at_end = ptr;

    while ( ptr != _at_front )
    {
        ilist_item *tmp = ptr->next();
        ptr->next( prev );
        prev = ptr;
        ptr = tmp;
    }
    _at_front->next( prev );
}
```

Вот реализация функции `reverse()`:

Тестовая программа для проверки этих операций выглядит так:

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

int main()
{
    ilist mylist;

    for ( int ix = 0; ix < 10; ++ix )
        { mylist.insert_front( ix ); }

    mylist.display();

    cout << "\n" << "инвертирование списка\n";
    mylist.reverse(); mylist.display();

    ilist mylist_too;
    mylist_too.insert_end(0); mylist_too.insert_end(1);
    mylist_too.insert_end(1); mylist_too.insert_end(2);
    mylist_too.insert_end(3); mylist_too.insert_end(5);

    cout << "\n" << "mylist_too:\n";
    mylist_too.display();

    mylist.concat( mylist_too );
    cout << "\n"
      << "mylist после concat с mylist_too:\n";
    mylist.disp1ay();

}
```

Результат работы программы:

```
( 10 ) ( 9 8 7 6 5 4 3 2 1 0 )

инвертирование списка

( 10 ) ( 0 1 2 3 4 5 6 7 8 9 )

mylist_too:
 ( 6 )( 0 1 1 2 3 5 )

mylist после concat с mylist_too:

( 16 ) ( 0 1 2 3 4 5 6 7 8 9 0 1 1 2 3 5 )
```

С одной стороны, задачу можно считать выполненной: мы не только реализовали все запланированные функции, но и проверили их работоспособность. С другой стороны, мы не обеспечили всех операций, которые необходимы для практического использования списка.

Одним из главных недостатков является то, что у пользователя нет способа перебирать элементы списка и он не может обойти это ограничение, поскольку реализация от него скрыта. Другим недостатком является отсутствие поддержки операций инициализации одного списка другим и присваивания одного списка другому. Мы сознательно не стали их реализовывать в первой версии, но теперь начнем улучшать наш класс.

Для реализации первой операции инициализации необходимо определить копирующий конструктор. Поведение такого конструктора, построенного компилятором по умолчанию, совершенно неправильно для нашего класса (как, собственно, и для любого класса, содержащего указатель в качестве члена), именно поэтому мы с самого начала

запретили его использование. Лучше уж полностью лишить пользователя какой-либо операции, чем допустить возможные ошибки. (В разделе 14.5 объясняется, почему действия копирующего конструктора по умолчанию в подобных случаях неверны.) Вот

```
ilist::ilist( const ilist &rhs )
{
    ilist_item *pt = rhs._at_front;
    while ( pt ) {
        insert_end( pt->value() );
        pt = pt->next();
    }
```

реализация конструктора, использующая функцию `insert_end()`:

```
}
```

Оператор присваивания должен сначала вызвать `remove_all()`, а затем с помощью `insert_end()` вставить все элементы второго списка. Поскольку эта операция

```
void ilist::insert_all ( const ilist &rhs )
{
    ilist_item *pt = rhs._at_front;
    while ( pt ) {
        insert_end( pt->value() );
        pt = pt->next();
    }
```

повторяется в обеих функциях, вынесем ее в отдельную функцию `insert_all()`:

```
}
```

```
inline ilist::ilist( const ilist &rhs )
    : _at_front( 0 ), _at_end( 0 )
    { insert_all ( rhs ); }

inline ilist&
ilist::operator=( const ilist &rhs ) {
    remove_all();
    insert_all( rhs );
    return *this;
```

после чего копирующий конструктор и оператор присваивания можно реализовать так:

```
}
```

Теперь осталось обеспечить пользователя возможностью путешествовать по списку, например с помощью доступа к члену `_at_front`:

```
ilist_item *ilist::front() { return _at_front(); }
```

После этого можно применить `ilist_item::next()`, как мы делали в функциях-членах:

```
ilist_item *pt = mylist.front();
while ( pt ) {
    do_something( pt->value() );
    pt = pt->next();
}
```

Хотя это решает проблему, лучше поступить иначе: реализовать общую концепцию прохода по элементам контейнера. В данном разделе мы расскажем об использовании

```
for ( ilist_item *iter = mylist.init_iter();
      iter;
      iter = mylist.next_iter() )
        do_something( iter->value() );
```

цикла такого вида:

(В разделе 2.8 мы уже касались понятия итератора. В главах 6 и 12 будут рассмотрены итераторы для имеющихся в стандартной библиотеке контейнерных типов и обобщенных алгоритмов.)

Наш итератор представляет собой несколько больше, чем просто указатель. Он должен уметь запоминать текущий элемент, возвращать следующий и определять, когда все элементы кончились. По умолчанию итератор инициализируется значением `_at_front`, однако пользователь может задать в качестве начального любой элемент списка. `next_iter()` возвращает следующий элемент или `0`, если элементов больше нет. Для

```
class ilist {
public:
    // ...
    init_iter( ilist_item *it = 0 );
private:
    //...
    ilist_item *_current;
};
```

реализации пришлось ввести дополнительный член класса:

```
inline ilist_item*
ilist::init_iter( i1ist_item *it )
{
    return _current = it ? it : _at_front;
}
```

`init_iter()` выглядит так:

`next_iter()` перемещает указатель `_current` на следующий элемент и возвращает его адрес, если элементы не кончились. В противном случае он возвращает `0` и устанавливает `_current` в `0`. Его реализацию можно представить следующим образом:

```
inline ilist_item*
ilist::
next_iter()
{
    ilist_item *next = _current
                     ? _current = _current->next()
                     : _current;
    return next;
}
```

Если элемент, на который указывает _current, удален, могут возникнуть проблемы. Их преодолевают модификацией кода функций remove() и remove_front(): они должны проверять значение _current. Если он указывает на удаляемый элемент, функции изменят его так, чтобы он адресовал следующий элемент либо был равен 0, когда удаляемый элемент – последний в списке или список стал пустым. Модифицированная

```
inline void
ilist::remove_front()
{
    if ( _at_front ) {
        ilist_item *ptr = _at_front;
        _at_front = _at_front->next();

        // _current не должен указывать на удаленный элемент
        if ( _current == ptr )
            _current = _at_front;

        bump_down_size();
        delete ptr;
    }
```

remove_front() выглядит так:

```
}
```

```
while ( plist ) {
    if ( plist->value() == value )
    {
        prev->next( plist->next() );
        if ( _current == plist )
```

Вот модифицированный фрагмент кода remove():

```
            _current = prev->next();
```

Что произойдет, если элемент будет вставлен перед тем, на который указывает _current? Значение _current не изменяется. Пользователь должен начать проход по списку с помощью вызова init_iter(), чтобы новый элемент попал в число перебираемых. При инициализации списка другим и при присваивании значение _current не копируется, а сбрасывается в 0.

Тестовая программа для проверки работы копирующего конструктора и оператора присваивания выглядит так::

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

int main()
{
    ilist mylist;

    for ( int ix = 0; ix < 10; ++ix ) {
        mylist.insert_front( ix );
        mylist.insert_end( ix );
    }

    cout << "\n" << "Применение init_iter() и next_iter() "
         << "для обхода всех элементов списка:\n";

    ilist_item *iter;
    for ( iter = mylist.init_iter();
          iter; iter = mylist.next_iter() )
        cout << iter->value() << " ";

    cout << "\n" << "Применение копирующего конструктора\n";

    ilist mylist2( mylist );
    mylist.remove_all();

    for ( iter = mylist2.init_iter();
          iter; iter = mylist2.next_iter() )
        cout << iter->value() << " ";

    cout << "\n" << "Применение копирующего оператора присваивания\n";
    mylist = mylist2;

    for ( iter = mylist.init_iter();
          iter; iter = mylist.next_iter() )
        cout << iter->value() << " ";

    cout << "\n";

}
```

Результат работы программы:

```
Применение init_iter() и next_iter() для обхода всех элементов списка:
9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
Применение копирующего конструктора
9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
Применение копирующего оператора присваивания
9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
```

5.11.1. Обобщенный список

Наш класс `ilist` имеет серьезный недостаток: он может хранить элементы только целого типа. Если бы он мог содержать элементы любого типа – как встроенного, так и определенного пользователем, – то его область применения была бы гораздо шире. Модифицировать `ilist` для поддержки произвольных типов данных позволяет механизм шаблонов (см. главу 16).

При использовании шаблона вместо параметра подставляется реальный тип данных. Например:

```
list< string > slist;
```

создает экземпляр списка, способного содержать объекты типа `string`, а

```
list< int > ilist;
```

создает список, в точности повторяющий наш `ilist`. С помощью шаблона класса можно обеспечить поддержку произвольных типов данных одним экземпляром кода. Рассмотрим последовательность действий, уделив особое внимание классу `list_item`.

Определение шаблона класса начинается ключевым словом `template`, затем следует список параметров в угловых скобках. Параметр представляет собой идентификатор,

```
template <class elemType>
```

перед которым стоит ключевое слово `class` или `typename`. Например:

```
class list_item;
```

Эта инструкция объявляет `list_item` шаблоном класса с единственным параметром-

```
template <typename elemType>
```

типом. Следующее объявление эквивалентно предыдущему:

```
class list_item;
```

Ключевые слова `class` и `typename` имеют одинаковое значение, можно использовать любое из них. Более удобное для запоминания `typename` появилось в стандарте C++ сравнительно недавно и поддерживается еще не всеми компиляторами. Поскольку наши тексты были написаны до появления этого ключевого слова, в них употребляется `class`. Шаблон класса `list_item` выглядит так:

```
template <class elemType>
class list_item {
public:
    list_item( elemType value, list_item *item = 0 )
             : _value( value ) {
        if ( !item )
            _next = 0;
        else {
            _next = item->_next;
            item->_next = this;
        }
    }

    elemType value() { return _value; }
    list_item* next() { return _next; }

    void next( list_item *link ) { _next = link; }
    void value( elemType new_value ) { _value = new_value; }

private:
    elemType   _value;
    list_item *_next;
};
```

Все упоминания типа `int` в определении класса `ilist_item` заменены на параметр `elemType`. Когда мы пишем:

```
list_item<doub1e> *ptr = new list_item<doub1e>( 3.14 );
```

компилятор подставляет `double` вместо `elemType` и создает экземпляр `list_item`, поддерживающий данный тип.

Аналогичным образом модифицируем класс `ilist` в шаблон класса `list`:

```
template <class elemType>
class list {
public:
list()
       : _at_front( 0 ), _at_end( 0 ), _current( 0 ),
         _size( 0 ) {}
    1ist( const list& );
    list& operator=( const list& );
    ~list() { remove_all(); }

    void insert ( list_item<elemType> *ptr, elemType value );
    void insert_end( elemType value );
    void insert_front( elemType value );
    void insert_all( const list &rhs );

    int remove( elemType value );
    void remove_front();
    void remove_all();

    list_item<elemType> *find( elemType value );
    list_item<elemType> *next_iter();
    list_item<elemType>* init_iter( list_item<elemType> *it );

    void disp1ay( ostream &os = cout );

    void concat( const list& );
    void reverse ();
    int size() { return _size; }

private:
    void bump_up_size()   { ++_size; }
    void bump_down_size() { --_size; }

    list_item<elemType> *_at_front;
    1ist_item<elemType> *_at_end;
    list_item<elemType> *_current;
    int _size;

};
```

Объекты шаблона класса `list` используются точно так же, как и объекты класса `ilist`. Основное преимущество шаблона в том, что он обеспечивает поддержку произвольных типов данных с помощью единственного определения.

(Шаблоны являются важной составной частью концепции программирования на C++. В главе 6 мы рассмотрим набор классов контейнерных типов, предоставляемых стандартной библиотекой C++. Неудивительно, что она содержит шаблон класса, реализующего операции со списками, равно как и шаблон класса, поддерживающего векторы; мы рассматривали их в главах 2 и 3.)

Наличие класса списка в стандартной библиотеке представляет некоторую проблему. Мы выбрали для нашей реализации название `list`, но, к сожалению, стандартный класс также носит это название. Теперь мы не можем использовать в программе одновременно оба класса. Конечно, проблему решит переименование нашего шаблона, однако во многих случаях эта возможность отсутствует.

Более общее решение состоит в использовании механизма пространства имен, который позволяет разработчику библиотеки заключить все свои имена в некоторое поименованное пространство и таким образом избежать конфликта с именами из глобального пространства. Применяя нотацию квалифицированного доступа, мы можем

употреблять эти имена в программах. Стандартная библиотека C++ помещает свои имена

```
namespace Primer_Third_Edition
{
    template <typename elemType>
    class list_item{ ... };

    template <typename elemType>
    class list{ ... };

    // ...
```

в пространство std. Мы тоже поместим наш код в собственное пространство:

```
}
```

Для использования такого класса в пользовательской программе необходимо написать

```
// наш заголовочный файл
#include "list.h"

// сделаем наши определения видимыми в программе
using namespace Primer_Third_Edition;

// теперь можно использовать наш класс list
list< int > ilist;
```

следующее:

```
// ...
```

(Пространства имен описываются в разделах 8.5 и 8.6.)

Упражнение 5.16

Мы не определили деструктор для ilist_item, хотя класс содержит указатель на динамическую область памяти. Причина заключается в том, что класс не выделяет память для объекта, адресуемого указателем _next, и, следовательно, не несет ответственности за ее освобождение. Начинающий программист мог бы допустить

```
ilist_item::~ilist_item()
{
    delete _next;
```

ошибку, вызвав деструктор для ilist_item:

```
}
```

Посмотрите на функции remove_all() и remove_front() и объясните, почему наличие такого деструктора является ошибочным.

Упражнение 5.17

```
void ilist::remove_end();
```

Наш класс ilist не поддерживает следующие операции:

```
void ilist::remove( ilist_item* );
```

Как вы думаете, почему мы их не включили? Реализуйте их.

Упражнение 5.18

Модифицируйте функцию `find()` так, чтобы вторым параметром она принимала адрес элемента, с которого нужно начинать поиск. Если этот параметр не задан, поиск начинается с первого элемента. (Поскольку мы добавляем второй параметр, имеющий значение по умолчанию, открытый интерфейс данной функции не меняется. Программы,

```
class ilist {
public:
    // ...
    ilist_item* find( int value, ilist_item *start_at = 0 );
    // ...
```

использующие предыдущую версию `find()`, будут работать без модификации.)

```
};
```

Упражнение 5.19

Используя новую версию `find()`, напишите функцию `count()`, которая подсчитывает количество вхождений элементов с заданным значением. Подготовьте тестовую программу.

Упражнение 5.20

Модифицируйте `insert(int value)` так, чтобы она возвращала указатель на вставленный объект `ilist_item`.

Упражнение 5.21

```
void ilist::
insert( ilist_item *begin,
        int *array_of_value,
```

Используя модифицированную версию `insert()`, напишите функцию:

```
        int elem_cnt );
```

где `array_of_value` указывает на массив значений, который нужно вставить в `ilist`, `elem_cnt` – на размер этого массива, а `begin` – на элемент, после которого производится вставка. Например, если есть `ilist`:

```
(3)( 0 1 21 )
```

и массив:

```
int ia[] = { 1, 2, 3, 5, 8, 13 };
```

вызов этой новой функции

```
ilist_item *it = mylist.find( 1 );
mylist.insert( it, ia, 6 );
```

изменит список таким образом:

```
(9) ( 0 1 1 2 3 5 8 13 21 )
```

Упражнение 5.22

Функции `concat()` и `reverse()` модифицируют оригинальный список. Это не всегда желательно. Напишите аналогичную пару функций, которые создают новый объект

```
ilist ilist::reverse_copy();
```

`ilist`:

```
ilist ilist::concat_copy( const ilist &rhs );
```

6. Абстрактные контейнерные типы

В этой главе мы продолжим рассмотрение типов данных, начатое в главе 3, представим дополнительную информацию о классах `vector` и `string` и познакомимся с другими контейнерными типами, входящими в состав стандартной библиотеки C++. Мы также расскажем об операторах и выражениях, упомянутых в главе 4, сосредоточив внимание на тех операциях, которые поддерживаются объектами контейнерных типов.

Последовательный контейнер содержит упорядоченный набор элементов одного типа. Можно выделить два основных типа контейнеров – вектор (`vector`) и список (`list`). (Третий последовательный контейнер – двусторонняя очередь (`deque`) – обеспечивает ту же функциональность, что и `vector`, но особенно эффективно реализует операции вставки и удаления первого элемента. `deque` следует применять, например, при реализации очереди, из которой извлекается только первый элемент. Все сказанное ниже относительно вектора применимо также и к `deque`.)

Ассоциативный контейнер эффективно реализует операции проверки существования и извлечения элемента. Два основных ассоциативных контейнера – это *отображение* (`map`) и *множество* (`set`). `map` состоит из пар ключ/значение, причем ключ используется для поиска элемента, а значение содержит хранимую информацию. Телефонный справочник хорошо иллюстрирует понятие отображения: ключом является фамилия и имя абонента, а значением – его телефонный номер.

Элемент контейнера `set` содержит только ключ, поэтому `set` эффективно реализует операцию проверки его существования. Этот контейнер можно применить, например, при реализации системы текстового поиска для хранения списка так называемых стоп-слов – слов, не используемых при поиске, таких, как *и*, *или*, *не*, *так* и тому подобных. Программа обработки текста считывает каждое слово и проверяет, есть ли оно в указанном списке. Если нет, то слово добавляется в базу данных.

В контейнерах `map` и `set` не может быть дубликатов – повторяющихся ключей. Для поддержки дубликатов существуют контейнеры `multimap` и `multiset`. Например, `multimap` можно использовать при реализации такого телефонного справочника, в котором содержится несколько номеров одного абонента.

В последующих разделах мы детально рассмотрим контейнерные типы и разработаем небольшую программу текстового поиска.

6.1. Система текстового поиска

В систему текстового поиска входят текстовый файл, указанный пользователем, и средство для задания запроса, состоящего из слов и, возможно, логических операторов.

Если одно или несколько слов запроса найдены, печатается количество их вхождений. По желанию пользователя печатаются предложения, содержащие найденные слова.

Например, если нужно найти все вхождения словосочетаний `Civil War` и `Civil Rights`, запрос может выглядеть таким образом[9]:

```
Civil && ( War || Rights )
```

Результат запроса:

```
Civil: 12 вхождений
War: 48 вхождений
Rights: 1 вхождение

Civil && War: 1 вхождение
Civil && Rights: 1 вхождение

(8) Civility, of course, is not to be confused with
Civil Rights, nor should it lead to Civil War
```

Здесь (8) представляет собой номер предложения в тексте. Наша система должна печатать фразы, содержащие найденные слова, в порядке возрастания их номеров (т.е. предложение номер 7 будет напечатано раньше предложения номер 9), не повторяя одну и ту же несколько раз.

Наша программа должна уметь:

- запросить имя текстового файла, а затем открыть и прочитать этот файл;
- организовать внутреннее представление этого файла так, чтобы впоследствии соотнести найденное слово с предложением, в котором оно встретилось, и определить порядковый номер этого слова ;
- понимать определенный язык запросов. В нашем случае он включает следующие операторы:

  `&&` два слова непосредственно следуют одно за другим в строке

  `||` одно или оба слова встречаются в строке

  `!` слово не встречается в строке

  `( )` группировка слов в запросе

Используя этот язык, можно написать:

```
Lincoln
```

чтобы найти все предложения, включающие слово *Lincoln*, или

---

9 Замечание. Для упрощения программы мы требуем, чтобы каждое слово было отделено пробелом от скобок и логических операторов. Таким образом, запросы вида

```
|(War || Rights)
Civil&&(War||Rights)
```

не будут поняты нашей системой. Хотя удобство пользователей не должно приноситься в жертву простоте реализации, мы считаем, что в данном случае можно смириться с таким ограничением.

```
! Lincoln
```

для поиска фраз, не содержащих такого слова, или же

```
( Abe || Abraham ) && Lincoln
```

для поиска тех предложений, где есть словосочетания *Abe Lincoln* или *Abraham Lincoln.*

Представим две версии нашей системы. В этой главе мы решим проблему чтения и хранения текстового файла в отображении, где ключом является слово, а значением – номер строки и позиции в строке. Мы обеспечим поиск по одному слову. (В главе 17 мы реализуем полную систему поиска, поддерживающую все указанные выше операторы языка запросов с помощью класса `Query`.) .

Возьмем шесть строчек из неопубликованного детского рассказа Стена Липпмана (Stan Lippman)10:

Рис. 2.

```
Alice Emma has long flowing red hair. Her Daddy says when the
wind blows through her hair, it looks almost alive, like a
fiery bird in flight. A beautiful fiery bird, he tells her,
magical but untamed. "Daddy, shush, there is no such thing,"
she tells him, at the same time wanting him to tell her more.
Shyly, she asks, "I mean. Daddy, is there?"
```

После считывания текста его внутреннее представление выглядит так (процесс считывания включает ввод очередной строки, разбиение ее на слова, исключение знаков препинания, замену прописных букв строчными, минимальная поддержка работы с суффиксами и исключение таких слов, как *and, a, the*):

```
alice ((0,0))
alive ((1,10))
almost ((1,9))
ask ((5,2))
beautiful ((2,7))
bird ((2,3),(2,9))
blow ((1,3))
daddy ((0,8),(3,3),(5,5))
emma ((0,1))
fiery ((2,2),(2,8))
flight ((2,5))
flowing ((0,4))
hair ((0,6),(1,6))
has ((0,2))
like ((2,0))
long ((0,3))
look ((1,8))
magical ((3,0))
mean ((5,4))
more ((4,12))
red ((0,5))
same ((4,5))
say ((0,9))
she ((4,0),(5,1))
shush ((3,4))
shyly ((5,0))
such ((3,8))
tell ((2,11),(4,1),(4,10))
```

10 Иллюстрация Елены Дрискилл (Elena Driskill).

```
there ((3,5),(5,7))
thing ((3,9))
through ((1,4))
time ((4,6))
untamed ((3,2))
wanting ((4,7))
wind ((1,2))
```

Ниже приводится пример работы программы, которая будет реализована в данном разделе (то, что задает пользователь, выделено курсивом):

```
please enter file name: alice_emma

enter a word against which to search the text.
to quit, enter a single character ==> alice

alice occurs 1 time:

    ( line 1 ) Alice Emma has long flowing red hair. Her Daddy says

enter a word against which to search the text.
to quit, enter a single character ==> daddy

daddy occurs 3 times:

    ( line 1 ) Alice Emma has long flow-ing red hair. Her Daddy says
    ( line 4 ) magical but untamed. "Daddy, shush, there is no such thing,"
    ( line 6 ) Shyly, she asks, "I mean, Daddy, is there?"

enter a word against which to search the text.
to quit, enter a single character ==> phoenix

Sorry. There are no entries for phoenix.

enter a word against which to search the text.
to quit, enter a single character ==> .
Ok, bye!
```

Для того чтобы реализация была достаточно простой, необходимо детально рассмотреть стандартные контейнерные типы и тип `string`, представленный в главе 3.

6.2. Вектор или список?

Первая задача, которую должна решить наша программа, – это считывание из файла заранее неизвестного количества слов. Слова хранятся в объектах типа `string`. Возникает вопрос: в каком контейнере мы будем хранить слова – в последовательном или ассоциативном?

С одной стороны, мы должны обеспечить возможность поиска слова и, в случае успеха, извлечь относящуюся к нему информацию. Отображение `map` является самым удобным для этого классом.

Но сначала нам нужно просто сохранить слова для предварительной обработки – исключения знаков препинания, суффиксов и т.п. Для этой цели последовательный контейнер подходит гораздо больше. Что же нам использовать: вектор или список?

Если вы уже писали программы на C или на C++ прежних версий, для вас, скорее всего, решающим фактором является возможность заранее узнать количество элементов. Если это количество известно на этапе компиляции, вы используете массив, в противном случае – список, выделяя память под очередной его элемент.

Однако это правило неприменимо к стандартным контейнерам: и `vector`, и `deque` допускают динамическое изменение размера. Выбор одного из этих трех классов должен

зависеть от способов, с помощью которых элементы добавляются в контейнер и извлекаются из него.

Вектор представляет собой область памяти, где элементы хранятся друг за другом. Для этого типа произвольный доступ (возможность извлечь, например, элемент 5, затем 15, затем 7 и т.д.) можно реализовать очень эффективно, поскольку каждый из них находится на некотором фиксированном расстоянии от начала. Однако вставка, кроме случая добавления в конец, крайне неэффективна: операция вставки в середину вектора потребует перемещения всего, что следует за вставляемым. Особенно это сказывается на больших векторах. (Класс deque устроен аналогично, однако операции вставки и удаления самого первого элемента работают в нем быстрее; это достигается двухуровневым представлением контейнера, при котором один уровень представляет собой реальное размещение элементов, а второй уровень адресует первый и последний из них.)

Список располагается в памяти произвольным образом. Каждый элемент содержит указатели на предыдущий и следующий, что позволяет перемещаться по списку вперед и назад. Вставка и удаление реализованы эффективно: изменяются только указатели. С другой стороны, произвольный доступ поддерживается плохо: чтобы прийти к определенному элементу, придется посетить все предшествующие. Кроме того, в отличие от вектора, дополнительно расходуется память под два указателя на каждый элемент списка.

Вот некоторые критерии для выбора одного из последовательных контейнеров:

- если требуется произвольный доступ к элементам, вектор предпочтительнее;
- если количество элементов известно заранее, также предпочтительнее вектор;
- если мы должны иметь возможность вставлять и удалять элементы в середину, предпочтительнее список;
- если нам не нужна возможность вставлять и удалять элементы в начало контейнера, вектор предпочтительнее, чем deque.

Как быть, если нам нужна возможность и произвольного доступа, и произвольного добавления/удаления элементов? Приходится выбирать: тратить время на поиск элемента или на его перемещение в случае вставки/удаления. В общем случае мы должны исходить из того, какую основную задачу решает приложение: поиск или добавление элементов? (Для выбора подхода может потребоваться измерение производительности для обоих типов контейнеров.) Если ни один из стандартных контейнеров не удовлетворяет нас, может быть, стоит разработать свою собственную, более сложную, структуру данных.

Какой из контейнеров выбрать, если мы не знаем количества его элементов (он будет динамически расти) и у нас нет необходимости ни в произвольном доступе, ни в добавлении элементов в середину? Что в таком случае более эффективно: список или вектор? (Мы отложим ответ на этот вопрос до следующего раздела.)

Список растет очень просто: добавление каждого нового элемента приводит к тому, что указатели на предыдущий и следующий для тех элементов, между которыми вставляется новый, меняют свои значения. В новом элементе таким указателям присваиваются значения адресов соседних элементов. Список использует только тот объем памяти, который нужен для имеющегося количества элементов. Накладными расходами являются два указателя в каждом элементе и необходимость использования указателя для получения значения элемента.

Внутреннее представление вектора и управление занимаемой им памятью более сложны. Мы рассмотрим это в следующем разделе.

Упражнение 6.1

Что лучше выбрать в следующих примерах: вектор, список или двустороннюю очередь? Или ни один из контейнеров не является предпочтительным?

1. Неизвестное заранее количество слов считывается из файла для генерации случайных предложений.
2. Считывается известное количество слов, которые вставляются в контейнер в алфавитном порядке.
3. Считывается неизвестное количество слов. Слова добавляются в конец контейнера, а удаляются всегда из начала.
4. Считывается неизвестное количество целых чисел. Числа сортируются и печатаются.

6.3. Как растет вектор?

Вектор может расти динамически. Как это происходит? Он должен выделить область памяти, достаточную для хранения всех элементов, скопировать в эту область все старые элементы и освободить ту память, в которой они содержались раньше. Если при этом элементы вектора являются объектами класса, то для каждого из них при таком копировании вызываются конструктор и деструктор. Поскольку у списка нет необходимости в таких дополнительных действиях при добавлении новых элементов, кажется очевидным, что ему проще поддерживать динамический рост контейнера. Однако на практике это не так. Давайте посмотрим почему.

Вектор может запрашивать память не под каждый новый элемент. Вместо этого она запрашивается с некоторым запасом, так что после очередного выделения вектор может поместить в себя некоторое количество элементов, не обращаясь за ней снова. (Каков размер этого запаса, зависит от реализации.) На практике такое свойство вектора обеспечивает значительное увеличение его эффективности, особенно для небольших объектов. Давайте рассмотрим некоторые примеры из реализации стандартной библиотеки C++ от компании Rogue Wave. Однако сначала определим разницу между размером и емкостью контейнера.

Емкость – это максимальное количество элементов, которое может вместить контейнер без дополнительного выделения памяти. (Емкостью обладают только те контейнеры, в которых элементы хранятся в непрерывной области памяти, – `vector`, `deque` и `string`. Для контейнера `list` это понятие не определено.) Емкость может быть получена с помощью функции `capacity()`. Размер – это реальное количество элементов, хранящихся в данный момент в контейнере. Размер можно получить с помощью функции `size()`. Например:

```
#include <vector>
#include <iostream>

int main()
{
    vector< int > ivec;
    cout << "ivec: размер: " << ivec.size()
         << " емкость: " << ivec.capacity() << endl;

    for ( int ix = 0; -ix < 24; ++ix ) {
        ivec.push_back( ix );
        cout << "ivec: размер: " << ivec.size()
             << " емкость: " << ivec.capacity() << endl;
    }

}
```

В реализации Rogue Wave и размер, и емкость ivec сразу после определения равны 0. После вставки первого элемента размер становится равным 1, а емкость – 256. Это значит, что до первого дополнительного выделения памяти в ivec можно вставить 256 элементов. При добавлении 256-го элемента вектор должен увеличиться: выделить память объемом в два раза больше текущей емкости, скопировать в нее старые элементы и освободить прежнюю память. Обратите внимание: чем больше и сложнее тип данных элементов, тем менее эффективен вектор в сравнении со списком. В таблице 6.1 показана зависимость начальной емкости вектора от используемого типа данных.

Таблица 6.1. Размер и емкость для различных типов данных

Тип данных	Размер в байтах	Емкость после первой вставки
int	4	256
double	8	128
простой класс #1	12	85
string	12	85
большой простой класс	8000	1
большой сложный класс	8000	1

Итак, в реализации Rogue Wave при первой вставке выделяется точно или примерно 1024 байта. После каждого дополнительного выделения памяти емкость удваивается. Для типа данных, имеющего большой размер, емкость мала, и увеличение памяти с копированием старых элементов происходит часто, вызывая потерю эффективности. (Говоря о сложных классах, мы имеем в виду класс, обладающий копирующим конструктором и операцией присваивания.) В таблице 6.2 показано время в секундах, необходимое для вставки десяти миллионов элементов разного типа в список и в вектор. Таблица 6.3 показывает время, требуемое для вставки 10 000 элементов (вставка элементов большего размера оказалась слишком медленной).

Таблица 6.2. Время в секундах для вставки 10 000 000 элементов

Тип данных	List	Vector
int	10.38	3.76

double	10.72	3.95
простой класс	12.31	5.89
string	14.42	11.80

Таблица 6.3. Время в секундах для вставки 10 000 элементов

Тип данных	List	Vector
большой простой класс	0.36	2.23
большой сложный класс	2.37	6.70

Отсюда следует, что вектор лучше подходит для типов данных малого размера, нежели список, и наоборот. Эта разница объясняется необходимостью выделения памяти и копирования в нее старых элементов. Однако размер данных – не единственный фактор, влияющий на эффективность. Сложность типа данных также ухудшает результат. Почему?

Вставка элемента как в список, так и в вектор, требует вызова копирующего конструктора, если он определен. (Копирующий конструктор инициализирует один объект значением другого. В разделе 2.2 приводится начальная информация, а в разделе 14.5 о таких конструкторах рассказывается подробно). Это и объясняет различие в поведении простых и сложных объектов при вставке в контейнер. Объекты простого класса вставляются побитовым копированием (биты одного объекта пересылаются в биты другого), а для строк и сложных классов это производится вызовом копирующего конструктора.

Вектор должен вызывать их для каждого элемента при перераспределении памяти. Более того, освобождение памяти требует работы деструкторов для всех элементов (понятие деструктора вводится в разделе 2.2). Чем чаще происходит перераспределение памяти, тем больше времени тратится на эти дополнительные вызовы конструкторов и деструкторов.

Конечно, одним из решений может быть переход от вектора к списку, когда эффективность вектора становится слишком низкой. Другое, более предпочтительное решение состоит в том, чтобы хранить в векторе не объекты сложного класса, а указатели на них. Такая замена позволяет уменьшить затраты времени на 10 000 вставок с 6.70 секунд до 0.82 секунды. Почему? Емкость возросла с 1 до 256, что существенно снизило частоту перераспределения памяти. Кроме того, копирующий конструктор и деструктор не вызываются больше для каждого элемента при копировании прежнего содержимого вектора.

Функция `reserve()` позволяет программисту явно задать емкость контейнера11.

```
int main() {
    vector< string > svec;
    svec.reserve( 32 ); // задает емкость равной 32
    // ...
```

Например:

11 Отметим, что deque не поддерживает операцию reserve()

```
}
```

svec получает емкость 32 при размере 0. Однако эксперименты показали, что любое изменение начальной емкости для вектора, у которого она по умолчанию отлична от 1, ведет к снижению производительности. Так, для векторов типа `string` и `double` увеличение емкости с помощью `reserve()` дало худшие показатели. С другой стороны, увеличение емкости для больших сложных типов дает значительный рост производительности, как показано в таблице 6.4.

Таблица 6.4. Время в секундах для вставки 10 000 элементов при различной емкости*

Емкость	Время в секундах
1 по умолчанию	670
4,096	555
8,192	444
10,000	222
*Сложный класс размером 8000 байт с конструктором копирования и деструктором	

В нашей системе текстового поиска для хранения объектов типа `string` мы будем использовать вектор, не меняя его емкости по умолчанию. Наши измерения показали, что производительность вектора в данном случае лучше, чем у списка. Но прежде чем приступать к реализации, посмотрим, как определяется объект контейнерного типа.

Упражнение 6.2

Объясните разницу между размером и емкостью контейнера. Почему понятие емкости необходимо для контейнера, содержащего элементы в непрерывной области памяти, и не нужно для списка?

Упражнение 6.3

Почему большие сложные объекты удобнее хранить в контейнере в виде указателей на них, а для коллекции целых чисел применение указателей снижает эффективность?

Упражнение 6.4

Объясните, какой из типов контейнера – вектор или список – больше подходит для приведенных примеров (во всех случаях происходит вставка неизвестного заранее числа элементов):.

(a) Целые числа

(b) Указатели на большие сложные объекты

(c) Большие сложные объекты

6.4. Как определить последовательный контейнер?

Для того чтобы определить объект контейнерного типа, необходимо сначала включить соответствующий заголовочный файл:

```
#include <vector>
#inclnde <list>
#include <deque>
#include <map>

#include <set>
```

Определение контейнера начинается именем его типа, за которым в угловых скобках

```
vector< string > svec;
```

следует тип данных его элементов[12]. Например:

```
list< int >      ilist;
```

Переменная svec определяется как вектор, способный содержать элементы типа string, а ilist – как список с элементами типа int. Оба контейнера при таком определении

```
if ( svec.empty() != true )
```

пусты. Чтобы убедиться в этом, можно вызвать функцию-член empty():

```
    ; // что-то не так
```

Простейший метод вставки элементов – использование функции-члена push_back(),

```
string text_word;
while ( cin >> text_word )
```

которая добавляет элементы в конец контейнера. Например:

```
     svec.push_back( text_word );
```

Здесь строки из стандартного ввода считываются в переменную text_word, и затем копия каждой строки добавляется в контейнер svec с помощью push_back().

Список имеет функцию-член push_front(), которая добавляет элемент в его начало. Пусть есть следующий массив:

```
int ia[ 4 ] = { 0, 1, 2, 3 };
```

---

12 Существующие на сегодняшний день реализации не поддерживают шаблоны с параметрами по умолчанию. Второй параметр – allocator – инкапсулирует способы выделения и освобождения памяти. В C++ он имеет значение по умолчанию, и его задавать не обязательно. Стандартная реализация использует операторы new и delete. Применение распределителя памяти преследует две цели: упростить реализацию контейнеров путем отделения всех деталей, касающихся работы с памятью, и позволить программисту при желании реализовать собственную стратегию выделения памяти. Определения объектов для компилятора, не поддерживающего значения по умолчанию параметров шаблонов, выглядят следующим образом:

```
vector< string, allocator > svec;
list< int, allocator >      ilist;
```

```
for ( int ix=0; ix<4; ++ix )
```

Использование push_back()

```
    ilist.push_back( ia[ ix ] );
```

```
for ( int ix=0; ix<4; ++ix )
```

создаст последовательность 0, 1, 2, 3, а push_front()

```
    ilist.push_front( ia[ ix ] );
```

создаст последовательность 3, 2, 1, 0. [13]

Мы можем при создании явно указать размер массива – как константным, так и

```
#include <list>
#include <vector>
#include <string>

extern int get_word_count( string file_name );
const int list_size = 64;

list< int > ilist( list_size );
```

неконстантным выражением:

```
vector< string > svec(get_word_count(string("Chimera")));
```

Каждый элемент контейнера инициализируется значением по умолчанию, соответствующим типу данных. Для int это 0. Для строкового типа вызывается конструктор по умолчанию класса string.

```
list< int > ilist( list_size, -1 );
```

Мы можем указать начальное значение всех элементов:

```
vector< string > svec( 24, "pooh" );
```

Разрешается не только задавать начальный размер контейнера, но и впоследствии изменять его с помощью функции-члена resize(). Например:

```
svec.resize( 2 * svec.size() );
```

Размер svec в этом примере удваивается. Каждый новый элемент получает значение по умолчанию. Если мы хотим инициализировать его каким-то другим значением, то оно указывается вторым параметром функции-члена resize():

---

13 Если функция-член push_front() используется часто, следует применять тип deque, а не vector: в deque эта операция реализована наиболее эффективно.

```
// каждый новый элемент получает значение "piglet"
svec.resize( 2 * svec.size(), "piglet" );
```

Кстати, какова наиболее вероятная емкость svec при определении, если его начальный размер равен 24? Правильно, 24! В общем случае минимальная емкость вектора равна его текущему размеру. При удвоении размера емкость, как правило, тоже удваивается

```
vector< string > svec2( svec );
```

Мы можем инициализировать новый контейнер с помощью существующего. Например:

```
list< int >      ilist2( ilist ) ;
```

Каждый контейнер поддерживает полный набор операций сравнения: равенство, неравенство, меньше, больше, меньше или равно, больше или равно. Сопоставляются попарно все элементы контейнера. Если они равны и размеры контейнеров одинаковы, то эти контейнеры равны; в противном случае – не равны. Результат операций “больше” или “меньше” определяется сравнением первых двух неравных элементов. Вот что печатает программа, сравнивающая пять векторов:

```
ivecl: 1 3 5 7 9 12
ivec2: 0 1 1 2 3 5 8 13
ivec3: 1 3 9
ivec4: 1 3 5 7
ivec5: 2 4

// первый неравный элемент: 1, 0
// ivecl больше чем ivec2
ivecl < ivec2 //false
ivec2 < ivecl //true

// первый неравный элемент: 5, 9
ivecl < ivec3 //true

// все элементы равны, но ivec4 содержит меньше элементов
// следовательно, ivec4 меньше, чем ivecl
ivecl < ivec4 //false

// первый неравный элемент: 1, 2
ivecl < ivec5 //true

ivecl == ivecl //true
ivecl == ivec4 //false
ivecl != ivec4 //true

ivecl > ivec2 //true
ivec3 > ivecl //true
ivec5 > ivec2 //true
```

Существуют три ограничения на тип элементов контейнера (практически это касается только пользовательских классов). Для должны быть определены:

- операция “равно”;
- операция “меньше” (все операции сравнения контейнеров, о которых говорилось выше, используют только эти две операции сравнения);
- значение по умолчанию (для класса это означает наличие конструктора по умолчанию).

Все предопределенные типы данных, включая указатели и классы из стандартной библиотеки C++ удовлетворяют этим требованиям.

Упражнение 6.5

```
#include <string>
#include <vector>
#include <iostream>

#int main()
{
    vector<string> svec;
    svec.reserve( 1024 );

    string text_word;
    while ( cin >> text_word )
        svec.push_back( text_word );

    svec.resize( svec.size()+svec.size()/2 );
    // ...
```

Объясните, что делает данная программа:

```
}
```

Упражнение 6.6

Может ли емкость контейнера быть меньше его размера? Желательно ли, чтобы емкость была равна размеру: изначально или после вставки элемента? Почему?

Упражнение 6.7

Если программа из упражнения 6.5 прочитает 256 слов, то какова наиболее вероятная емкость контейнера после изменения размера? А если она считает 512 слов? 1000? 1048?

Упражнение 6.8

Какие из данных классов не могут храниться в векторе:

```
(a) class cl1 {
public:
    c11( int=0 );
    bool operator==();
    bool operator!=();
    bool operator<=();
    bool operator<();
// ...
};

(b) class c12 {
public:
    c12( int=0 );
    bool operator!=();
    bool operator<=();
    // ...
};

(c) class c13 {
public:
    int ival;
};

(d) class c14 {
public:
    c14( int, int=0 );
    bool operator==();
    bool operator!=();
    // ...

}
```

6.5. Итераторы

Итератор предоставляет обобщенный способ перебора элементов любого контейнера – как последовательного, так и ассоциативного. Пусть `iter` является итератором для какого-либо контейнера. Тогда

```
++iter;
```

перемещает итератор так, что он указывает на следующий элемент контейнера, а

```
*iter;
```

разыменовывает итератор, возвращая элемент, на который он указывает.

Все контейнеры имеют функции-члены `begin()` и `end()`.

- `begin()` возвращает итератор, указывающий на первый элемент контейнера.
- `end()` возвращает итератор, указывающий на элемент, следующий за последним в контейнере.

```
for ( iter = container. begin();
      iter != container.end(); ++iter )
```

Чтобы перебрать все элементы контейнера, нужно написать:

```
    do_something_with_element( *iter );
```

Объявление итератора выглядит слишком сложным. Вот определение пары итераторов

```
// vector<string> vec;
vector<string>::iterator iter = vec.begin();
```

вектора типа `string`:

```
vector<string>::iterator iter_end = vec.end();
```

В классе `vector` для определения `iterator` используется `typedef`. Синтаксис

```
vector<string>::iterator
```

ссылается на `iterator`, определенный с помощью `typedef` внутри класса `vector`, содержащего элементы типа `string`.

```
for( ; iter != iter_end; ++iter )
```

Для того чтобы напечатать все элементы вектора, нужно написать:

```
    cout << *iter << '\n';
```

Здесь значением `*iter` выражения является, конечно, элемент вектора.

В дополнение к типу `iterator` в каждом контейнере определен тип `const_iterator`, который необходим для навигации по контейнеру, объявленному как `const`.

```
#include <vector>
void even_odd( const vector<int> *pvec,
               vector<int> *pvec_even,
               vector<int> *pvec_odd )
{
    // const_iterator необходим для навигации по pvec
    vector<int>::const_iterator c_iter = pvec->begin();
    vector<int>::const_1terator c_iter_end = pvec->end();

    for ( ; c_iter != c_iter_end; ++c_iter )
        if ( *c_iter % 2 )
            pvec_even->push_back( *c_iter );
        else pvec_odd->push_back( *c_iter );
```

`const_iterator` позволяет только читать элементы контейнера:

```
}
```

Что делать, если мы хотим просмотреть некоторое подмножество элементов, например взять каждый второй или третий элемент, или хотим начать с середины? Итераторы поддерживают адресную арифметику, а значит, мы можем прибавить некоторое число к итератору:

```
vector<int>::iterator iter = vec->begin()+vec.size()/2;
```

`iter` получает значение адреса элемента из середины вектора, а выражение

```
iter += 2;
```

сдвигает iter на два элемента.

Арифметические действия с итераторами возможны только для контейнеров vector и deque. list не поддерживает адресную арифметику, поскольку его элементы не располагаются в непрерывной области памяти. Следующее выражение к списку неприменимо:

```
ilist.begin() + 2;
```

так как для перемещения на два элемента необходимо два раза перейти по адресу, содержащемуся в закрытом члене next. У классов vector и deque перемещение на два элемента означает прибавление 2 к указателю на текущий элемент. (Адресная арифметика рассматривается в разделе 3.3.)

Объект контейнерного типа может быть инициализирован парой итераторов, обозначающих начало и конец последовательности копируемых в новый объект элементов. (Второй итератор должен указывать на элемент, следующий за последним

```
#include <vector>
#include <string>
#include <iostream>

int main()
{
    vector<string> svec;
    string intext;
    while ( cin >> intext )
        svec.push_back( intext );

    // обработать svec ...
```

копируемым.) Допустим, есть вектор:

```
}
```

Вот как можно определить новые векторы, инициализируя их элементами первого

```
int main() {
    vector<string> svec;
    // ...

    // инициализация svec2 всеми элементами svec
    vector<string> svec2( svec.begin(), svec.end() );

    // инициализация svec3 первой половиной svec
    vector<string>::iterator it =
              svec.begin() + svec.size()/2;
    vector<string> svec3 ( svec.begin(), it );

    // ...
```

вектора:

```
}
```

Использование специального типа istream_iterator (о нем рассказывается в разделе

```
#include <vector>
#include <string>
#include <iterator>

int mainQ
{
    // привязка istream_iterator к стандартному вводу
    istream_iterator<string> infile( cin );

    // istream_iterator, отмечающий конец потока
    istream_iterator<string> eos;

    // инициализация svec элементами, считываемыми из cin;
    vector<string> svec( infile, eos );

    // ...
```

12.4.3) упрощает чтение элементов из входного потока в svec:

```
}
```

Кроме итераторов, для задания диапазона значений, инициализирующих контейнер, можно использовать два указателя на массив встроенного типа. Пусть есть следующий

```
#include <string>
string words[4] = {
    "stately", "plump", "buck", "mulligan"
```

массив строк:

```
};
```

Мы можем инициализировать вектор с помощью указателей на первый элемент массива и на элемент, следующий за последним:

```
vector< string > vwords( words, words+4 );
```

Второй указатель служит “стражем”: элемент, на который он указывает, не копируется.

```
int ia[6] = { 0, 1, 2, 3, 4, 5 };
```

Аналогичным образом можно инициализировать список целых элементов:

```
list< int > ilist( ia, ia+6 );
```

В разделе 12.4 мы снова обратимся к итераторам и опишем их более детально. Сейчас информации достаточно для того, чтобы использовать итераторы в нашей системе текстового поиска. Но прежде чем вернуться к ней, рассмотрим некоторые дополнительные операции, поддерживаемые контейнерами.

Упражнение 6.9

Какие ошибки допущены при использовании итераторов:

```
const vector< int > ivec;
vector< string >    svec;
list< int >         ilist;

(a) vector<int>::iterator it = ivec.begin();
(b) list<int>::iterator   it = ilist.begin()+2;
(c) vector<string>::iterator it = &svec[0];
(d) for ( vector<string>::iterator
           it = svec.begin(); it != 0; ++it )

    // ...
```

Упражнение 6.10

```
int ia[7] = { 0, 1, 1, 2, 3, 5, 8 };
string sa[6] = {
    "Fort Sumter", "Manassas", "Perryville", "Vicksburg",
    "Meridian", "Chancellorsvine" };

(a) vector<string> svec( sa, &sa[6] );
(b) list<int> ilist( ia+4, ia+6 );
(c) list<int> ilist2( ilist.begin(), ilist.begin()+2 );
(d) vector<int> ivec( &ia[0], ia+8 );
(e) list<string> slist( sa+6, sa );
```

Найдите ошибки в использовании итераторов:

```
(f) vector<string> svec2( sa, sa+6 );
```

6.6. Операции с последовательными контейнерами

Функция-член `push_back()` позволяет добавить единственный элемент в конец контейнера. Но как вставить элемент в произвольную позицию? А целую последовательность элементов? Для этих случаев существуют более общие операции.

```
vector< string > svec;
list< string > slist;
string spouse( "Beth" );

slist.insert( slist.begin(), spouse );
```

Например, для вставки элемента в начало контейнера можно использовать:

```
svec.insert( svec.begin(), spouse );
```

Первый параметр функции-члена `insert()` (итератор, адресующий некоторый элемент контейнера) задает позицию, а второй – вставляемое перед этой позицией значение. В примере выше элемент добавляется в начало контейнера. А так можно реализовать вставку в произвольную позицию:

```
string son( "Danny" );

list<string>::iterator iter;
iter = find( slist.begin(), slist.end(), son );

slist.insert( iter, spouse );
```

Здесь find() возвращает позицию элемента в контейнере, если элемент найден, либо итератор end(), если ничего не найдено. (Мы вернемся к функции find() в конце следующего раздела.) Как можно догадаться, push_back() эквивалентен следующей

```
// эквивалентный вызов: slist.push_back( value );
```

записи:

```
slist.insert( slist.end(), value );
```

Вторая форма функции-члена insert() позволяет вставить указанное количество одинаковых элементов, начиная с определенной позиции. Например, если мы хотим

```
vector<string> svec;
string anna( "Anna" );
```

добавить десять элементов Anna в начало вектора, то должны написать:

```
svec.insert( svec.begin(), 10, anna );
```

insert() имеет и третью форму, помогающую вставить в контейнер несколько элементов. Допустим, имеется следующий массив:

```
string sarray[4] = { "quasi", "simba", "frollo", "scar" };
```

Мы можем добавить все его элементы или только некоторый диапазон в наш вектор

```
svec.insert( svec.begin(), sarray, sarray+4 );
svec.insert( svec.begin() + svec.size()/2,
```

строк:

```
     sarray+2, sarray+4 );
```

```
// вставляем элементы svec
// в середину svec_two
svec_two.insert( svec_two.begin() + svec_two.size()/2,
```

Такой диапазон отмечается и с помощью пары итераторов

```
                  svec.begin(), svec.end() );
```

```
list< string > slist;

// ...

// вставляем элементы svec
// перед элементом, содержащим stringVal
list< string >::iterator iter =
    find( slist.begin(), slist.end(), stringVal );
```

или любого контейнера, содержащего строки:[14]

```
slist.insert( iter, svec.begin(), svec.end() );
```

6.6.1. Удаление

В общем случае удаление осуществляется двумя формами функции-члена `erase()`. Первая форма удаляет единственный элемент, вторая – диапазон, отмеченный парой итераторов. Для последнего элемента можно воспользоваться функцией-членом `pop_back()`.

При вызове `erase()` параметром является итератор, указывающий на нужный элемент. В следующем фрагменте кода мы воспользуемся обобщенным алгоритмом `find()` для

```
string searchValue( "Quasimodo" );
list< string >::iterator iter =
    find( slist.begin(), slist.end(), searchValue );
if ( iter != slist.end() )
```

нахождения элемента и, если он найден, передадим его адрес функции-члену `erase()`.

```
    slist.erase( iter );
```

Для удаления всех элементов контейнера или некоторого диапазона можно написать

```
// удаляем все элементы контейнера
slist.erase( slist.begin(), slist.end() );

// удаляем элементы, помеченные итераторами
list< string >::iterator first, last;

first = find( slist. begin(), slist.end(), vail );
last = find( slist.begin(), slist.end(), va12 );

// ... проверка first и last
```

следующее:

```
slist.erase( first, last );
```

---

14 Последняя форма `insert()` требует, чтобы компилятор работал с шаблонами функций-членов. Если ваш компилятор еще не поддерживает это свойство стандарта C++, то оба контейнера должны быть одного типа, например два списка или два вектора, содержащих элементы одного типа.

Парной по отношению к push_back() является функция-член pop_back(), удаляющая

```
vector< string >::iterator iter = buffer.begin();
for ( ; iter != buffer.end(), iter++ )
{
    slist.push_back( *iter );
    if ( ! do_something( slist ))
        slist.pop_back();
```

из контейнера последний элемент, не возвращая его значения:

```
}
```

6.6.2. Присваивание и обмен

Что происходит, если мы присваиваем один контейнер другому? Оператор присваивания копирует элементы из контейнера, стоящего справа, в контейнер, стоящий слева от знака

```
// svecl содержит 10 элементов
// svec2 содержит 24 элемента
// после присваивания оба содержат по 24 элемента
```

равенства. А если эти контейнеры имеют разный размер? Например:

```
svecl = svec2;
```

Контейнер-адресат (svec1) теперь содержит столько же элементов, сколько контейнер-источник (svec2). 10 элементов, изначально содержавшихся в svec1, удаляются (для каждого из них вызывается деструктор класса string).

Функция обмена swap() может рассматриваться как дополнение к операции присваивания. Когда мы пишем:

```
svecl.swap( svec2 );
```

svec1 после вызова функции содержит 24 элемента, которые он получил бы в результате присваивания:

```
svecl = svec2;
```

но зато теперь svec2 получает 10 элементов, ранее находившихся в svec1. Контейнеры "обмениваются" своим содержимым.

6.6.3. Обобщенные алгоритмы

Операции, описанные в предыдущих разделах, составляют набор, поддерживаемый непосредственно контейнерами vector и deque. Согласитесь, что это весьма небогатый интерфейс и ему явно не хватает базовых операций find(), sort(), merge() и т.д. Планировалось вынести общие для всех контейнеров операции в набор обобщенных алгоритмов, которые могут применяться ко всем контейнерным типам, а также к массивам встроенных типов. (Обобщенные алгоритмы описываются в главе 12 и в Приложении.) Эти алгоритмы связываются с определенным типом контейнера с

помощью передачи им в качестве параметров пары соответствующих итераторов. Вот как

```
#include <list>
#include <vector>

int ia[ 6 ] = { 0, 1, 2, 3, 4, 5 };
vector<string> svec;
list<double> dtist;

// соответствующий заголовочный файл
#include <algorithm>
vector<string>::iterator viter;
list<double>::iterator liter;
#int *pia;

// find() возвращает итератор на найденный элемент
// для массива возвращается указатель ...
pia =   find( &ia[0], &ia[6], some_int_value );
liter = find( dlist.begin(), dlist.end(), some_double_value );
```

выглядят вызовы алгоритма `find()` для списка, вектора и массива разных типов:

```
viter = find( svec.begin(), svec.end(), some_string_value );
```

Контейнер `list` поддерживает дополнительные операции, такие, как `sort()` и `merge()`, поскольку в нем не реализован произвольный доступ к элементам. (Эти операции описаны в разделе 12.6.)

Теперь вернемся к нашей поисковой системе.

Упражнение 6.11

```
int ia[] = { 1, 5, 34 };
int ia2[] = { 1, 2, 3 };
int ia3[] = { 6, 13, 21, 29, 38, 55, 67, 89 };
```

Напишите программу, в которой определены следующие объекты:

```
vector<int> ivec;
```

Используя различные операции вставки и подходящие значения `ia`, `ia2` и `ia3`, модифицируйте вектор `ivec` так, чтобы он содержал последовательность:

```
{ 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 }
```

Упражнение 6.12

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 };
```

Напишите программу, определяющую данные объекты:

```
list<int> ilist( ia, ia+11 );
```

Используя функцию-член `erase()` с одним параметром, удалите из `ilist` все нечетные элементы.

6.7. Читаем текстовый файл

Первая наша задача – прочитать текстовый файл, в котором будет производиться поиск. Нам нужно сохранить следующую информацию: само слово, номер строки и позицию в строке, где слово встречается.

Как получить одну строку текста? Стандартная библиотека предоставляет для этого

```
istream&
```

функцию `getline()`:

```
getline( istream &is, string str, char delimiter );
```

`getline()`берет из входного потока все символы, включая пробелы, и помещает их в объект типа `string`, до тех пор пока не встретится символ `delimiter`, не будет достигнут конец файла или количество полученных символов не станет равным величине, возвращаемой функцией-членом `max_size()`класса `string`.

Мы будем помещать каждую такую строку в вектор.

Мы вынесли код, читающий файл, в функцию, названную `retrieve_text()`. В объекте типа `pair` дополнительно сохраняется размер и номер самой длинной строки. (Полный текст программы приводится в разделе 6.14.)

Вот реализация функции ввода файла:[15]

---

15 Программа компилировалась компилятором, не поддерживающим значений параметров по умолчанию шаблонов. Поэтому нам пришлось явно указать аллокатор:

```
vector<string,allocator> *lines_of_text;
```

Для компилятора, полностью соответствующего стандарту C++, достаточно отметить тип элементов:

```
vector<string> *lines_of_text;
```

```
// возвращаемое значение - указатель на строковый вектор
vector<string,allocator>*
retrieve_text()
{
    string file_name;

    cout << "please enter file name: ";
    cin >> file_name;

    // откроем файл для ввода ...
    ifstream 1nfile( file_name.c_str(), ios::in );
    if ( ! infile ) {
        cerr << "oops! unable to open file "
             << file_name << " -- bailing out!\n";
        exit( -1 );
    }
    else cout << '\n';

    vector<string, allocator> *1ines_of_text =
        new vector<string, allocator>;
    string textime;

    typedef pair<string::size_type, int> stats;
    stats maxline;
    int   linenum = 0;

    while ( getline( infile, textline, '\n' )) {
        cout << "line read: " << textline << '\n';

        if ( maxline.first < textline.size() ) {
            maxline.first = textline.size() ;
            maxline.second = linenum;
        }

        1ines_of_text->push_back( textline );
        linenum++;
    }

    return lines_of_text;

}
```

Вот как выглядит вывод программы (размер страницы книги недостаточен, чтобы расположить напечатанные строки во всю длину, поэтому мы сделали в тексте отступы, показывающие, где реально заканчивалась строка):

```
please enter file name: a1ice_emma

line read: Alice Emma has long flowing red hair. Her Daddy says
line read: when the wind blows through her hair, it looks
           almost alive,
line read: like a fiery bird in flight. A beautiful fiery bird,
           he tells her,
line read: magical but untamed. "Daddy, shush, there is no such
           thing, "
line read: she tells him, at the same time wanting him to tell
           her more.
line read: Shyly, she asks, "I mean. Daddy, is there?"

number of lines: 6
maximum length: 66
longest line: like a fiery bird in flight. A beautiful fiery
              bird, he tells her,
```

После того как все строки текста сохранены, нужно разбить их на слова. Сначала мы отбросим знаки препинания. Например, возьмем строку из части “Anna Livia Plurrabelle” романа “Finnegans Wake”.

```
"For every tale there's a telling,
and that's the he and she of it."
```

В приведенном фрагменте есть следующие знаки препинания:

```
"For
there's
telling,
that's
it."
```

А хотелось бы получить:

```
For
there
telling
that
it
```

Можно возразить, что

```
there's
```

должно превратиться в

```
there is
```

но мы-то движемся в другом направлении: следующий шаг – это отбрасывание семантически нейтральных слов, таких, как *is, that, and, it* и т.д. Так что для данной строчки из “Finnegans Wake” только два слова являются значимыми: *tale* и *telling*, и только по этим словам будет выполняться поиск. (Мы реализуем набор стоп-слов с помощью контейнерного типа `set`, который подробно рассматривается в следующем разделе.)

После удаления знаков препинания нам необходимо превратить все прописные буквы в строчные, чтобы избежать проблем с поиском в таких, например, строках:

```
Home is where the heart is.
A home is where they have to let you in.
```

Несомненно, запрос слова `home` должен найти обе строки.

Мы должны также обеспечить минимальную поддержку учета словоформ: отбрасывать окончания слов, чтобы слова *dog* и *dogs*, *love*, *loving* и *loved* рассматривались системой как одинаковые.

В следующем разделе мы вернемся к описанию стандартного класса string и рассмотрим многочисленные операции над строками, которые он поддерживает, в контексте дальнейшей разработки нашей поисковой системы.

6.8. Выделяем слова в строке

Нашей первой задачей является разбиение строки на слова. Мы будем вычленять слова, находя разделяющие их пробелы с помощью функции find(). Например, в строке

```
Alice Emma has long flowing red hair.
```

насчитывается шесть пробелов, следовательно, эта строка содержит семь слов.

Класс string имеет несколько функций поиска. find() – наиболее простая из них. Она ищет образец, заданный как параметр, и возвращает позицию его первого символа в строке, если он найден, или специальное значение string::npos в противном случае.

```
#include <string>
#include <iostream>

int main() {
    string name( "AnnaBelle" );
    int pos = name.find( "Anna" );
    if ( pos == string::npos )
        cout << "Anna не найдено!\n";
    else cout << "Anna найдено в позиции: " << pos << endl;
}
```

Например:

Хотя позиция подстроки почти всегда имеет тип int, более правильное и переносимое объявление типа результата, возвращаемого find(), таково:

```
string::size_type
```

Например:

```
string::size_type pos = name.find( "Anna" );
```

Функция find() делает не совсем то, что нам надо. Требуемая функциональность обеспечивается функцией find_first_of(), которая возвращает позицию первого символа, соответствующего одному из заданных в строке-параметре. Вот как найти первый символ, являющийся цифрой:

```
#include <string>
#include <iostream>

int main() {
    string numerics( "0123456789" );
    string name( "r2d2" );

    string:: size_type pos = name.find_first_of( numerics );
    cout << "найдена цифра в позиции: "
         << pos       << "\tэлемент равен "
         << name[pos] << endl;
}
```

В этом примере pos получает значение 1 (напоминаем, что символы строки нумеруются с 0).

Но нам нужно найти все вхождения символа, а не только первое. Такая возможность реализуется передачей функции find_first_of() второго параметра, указывающего позицию, с которой начать поиск. Изменим предыдущий пример. Можете ли вы сказать,

```
#include <string>
#include <iostream>
int main() {
    string numerics( "0123456789" );
    string name( "r2d2" );

    string::size_type pos = 0;

    // где-то здесь ошибка!
    while (( pos = name.find_first_of( numerics, pos ))
                != string::npos )
        cout << "найдена цифра в позиции: "
             << pos       << "\tэлемент равен "
             << name[pos] << endl;
}
```

что в нем все еще не вполне удовлетворительно?

В начале цикла pos равно 0, поэтому поиск идет с начала строки. Первое вхождение обнаружено в позиции 1. Поскольку найденное значение не совпадает с string::npos, выполнение цикла продолжается. Для второго вызова find_first_of()значение pos равно 1. Поиск начнется с 1-й позиции. Вот ошибка! Функция find_first_of() снова найдет цифру в первой позиции, и снова, и снова... Получился бесконечный цикл. Нам необходимо увеличивать pos на 1 в конце каждой итерации:

```
// исправленная версия цикла
while (( pos = name.find_first_of( numerics, pos ))
            != string::npos )
{
        cout << "найдена цифра в позиции: "
             << pos       << "\tэлемент равен "
             << name[pos] << endl;

    // сдвинуться на 1 символ
    ++pos;

}
```

Чтобы найти все пустые символы (к которым, помимо пробела, относятся символы табуляции и перевода строки), нужно заменить строку `numerics` в этом примере строкой, содержащей все эти символы. Если же мы уверены, что используется только символ

```
// фрагмент программы
while (( pos = textline.find_first_of( ' ', pos ))
            != string::npos )
```

пробела и никаких других, то можем явно задать его в качестве параметра функции:

```
    // ...
```

```
// фрагмент программы
// pos: позиция на 1 большая конца слова
// prev_pos: позиция начала слова

string::size_type pos = 0, prev_pos = 0;

while (( pos = textline.find_first_of( ' ', pos ))
            != string::npos )
{
    // ...
    // запомнить позицию начала слова
    prev_pos = ++pos;
}
```

Чтобы узнать длину слова, введем еще одну переменную:

На каждой итерации `prev_pos` указывает позицию начала слова, а `pos` – позицию следующего символа после его конца. Соответственно, длина слова равна:

```
pos - prev_pos; // длина слова
```

После того как мы выделили слово, необходимо поместить его в строковый вектор. Это можно сделать, копируя в цикле символы из `textline` с позиции `prev_pos` до `pos -1`. Функция `substr()` сделает это за нас:

```
// фрагмент программы
vector<string> words;

while (( pos = textline.find_first_of( ' ', pos ))
            != string::npos )
{
    words.push_back( textline.substr(
                    prev_pos, pos-prev_pos));
    prev_pos = ++pos;
}
```

Функция `substr()` возвращает копию подстроки. Первый ее аргумент обозначает первую позицию, второй – длину подстроки. (Второй аргумент можно опустить, тогда подстрока включит в себя остаток исходной строки, начиная с указанной позиции.)

В нашей реализации допущена ошибка: последнее слово не будет помещено в контейнер. Почему? Возьмем строку:

```
seaspawn and seawrack
```

После каждого из первых двух слов поставлен пробел. Два вызова функции `find_first_of()` вернут позиции этих пробелов. Третий же вызов вернет `string::npos`, и цикл закончится. Таким образом, последнее слово останется необработанным.

Вот полный текст функции, названной нами `separate_words()`. Помимо сохранения слов в векторе строк, она вычисляет координаты каждого слова – номер строки и колонки (нам эта информация потребуется впоследствии).

```
typedef pair<short,short> location;
typedef vector<location>  loc;
typedef vector<string>    text;
typedef pair<text* ,loc*> text_loc;

text_loc*
separate_words( const vector<string> *text_file )
{
    // words: содержит набор слов
    // locations: содержит информацию о строке и позиции
    // каждого слова
    vector<string>   *words = new vector<string>;
    vector<location> * locations = new vector<location>;

    short line_pos = 0; // текущий номер строки
    // iterate through each line of text
    for ( ; line_pos < text_file->size(); ++line_pos )
        // textline: обрабатываемая строка
        // word_pos: позиция в строке
        short word_pos = 0;
        string textline = (*text_file) [ line_pos ];

        string::size_type pos = 0, prev_pos = 0;

        while (( pos = textline.find_first_of( ' ', pos ))
                    != string::npos )
        {
            // сохраним слово
            words->push_back(
                textline.substr( prev_pos, pos - prev_pos ));

            // сохраним информацию о его строке и позиции
            locations->push_back(
                make_pair( line_pos, word_pos ));

            // сместим позицию для следующей итерации
            ++word_pos; prev_pos = ++pos;
        }

        // обработаем последнее слово
        words->push_back(
               textline.substr( prev_pos, pos - prev_pos ));

        locations->push_back(
               make_pair( line_pos, word_pos ));
    }
    return new text_loc( words, locations );
}
```

```
int main()
{
    vector<string> *text_file = retrieve_text();
    text_loc *text_locations = separate_words( text_file );
    // ...
```

Теперь функция `main()` выглядит следующим образом:

```
}
```

Вот часть распечатки, выданной тестовой версией separate_words():

```
textline: Alice Emma has long flowing red hair. Her Daddy
          says

eol: 52 pos: 5 line: 0 word: 0 substring: Alice
eol: 52 pos: 10 line: 0 word: 1 substring: Emma
eol: 52 pos: 14 line: 0 word: 2 substring: has
eol: 52 pos: 19 line: 0 word: 3 substring: long
eol: 52 pos: 27 line: 0 word: 4 substring: flowing
eol: 52 pos: 31 line: 0 word: 5 substring: red
eol: 52 pos: 37 line: 0 word: 6 substring: hair.
eol: 52 pos: 41 line: 0 word: 7 substring: Her
eol: 52 pos: 47 line: 0 word: 8 substring: Daddy
last word on line substring: says

...

textline: magical but untamed. "Daddy, shush, there is no
          such thing,"

eol: 60 pos: 7 line: 3 word: 0 substring: magical
eol: 60 pos: 11 line: 3 word: 1 substring: but
eol: 60 pos: 20 line: 3 word: 2 substring: untamed
eol: 60 pos: 28 line: 3 word: 3 substring: "Daddy,
eol: 60 pos: 35 line: 3 word: 4 substring: shush,
eol: 60 pos: 41 line: 3 word: 5 substring: there
eol: 60 pos: 44 line: 3 word: 6 substring: is
eol: 60 pos: 47 line: 3 word: 7 substring: no
eol: 60 pos: 52 line: 3 word: 8 substring: such
last word on line substring: thing,":

...


textline: Shy1y, she asks, "I mean, Daddy: is there?"

eol: 43 pos: 6 line: 5 word: 0 substring: Shyly,
eol: 43 pos: 10 line: 5 word: 1 substring: she
eol: 43 pos: 16 line: 5 word: 2 substring: asks,
eol: 43 pos: 19 line: 5 word: 3 substring: "I
eol: 43 pos: 25 line: 5 word: 4 substring: mean,
eol: 43 pos: 32 line: 5 word: 5 substring: Daddy,
eol: 43 pos: 35 line: 5 word: 6 substring: is
last word on line substring: there?":
```

Прежде чем продолжить реализацию поисковой системы, вкратце рассмотрим оставшиеся функции-члены класса string, предназначенные для поиска. Функция

```
string river( "Mississippi" );
string::size_type first_pos = river.find( "is" );
```

rfind() ищет последнее, т.е. самое правое, вхождение указанной подстроки:

```
string::size_type 1ast_pos = river.rfind( "is" );
```

find() вернет 1, указывая позицию первого вхождения подстроки "is", а rfind() – 4 (позиция последнего вхождения "is").

find_first_not_of() ищет первый символ, не содержащийся в строке, переданной как параметр. Например, чтобы найти первый символ, не являющийся цифрой, можно написать:

```
string elems( "0123456789" );
string dept_code( "03714p3" );

// возвращается позиция символа 'p'
string::size_type pos = dept_code.find_first_not_of(elems) ;
```

`find_last_of()` ищет последнее вхождение одного из указанных символов. `find_last_not_of()` – последний символ, не совпадающий ни с одним из заданных. Все эти функции имеют второй необязательный параметр – позицию в исходной строке, с которой начинается поиск.

Упражнение 6.13

Напишите программу, которая ищет в строке

```
"ab2c3d7R4E6"
```

цифры, а затем буквы, используя сначала `find_first_of()`, а потом `find_first_not_of()`.

Упражнение 6.14

Напишите программу, которая подсчитывает все слова и определяет самое длинное и

```
string linel = "We were her pride of 10 she named us --";
string line2 = "Benjamin, Phoenix, the Prodigal"
string line3 = "and perspicacious pacific Suzanne";
```

самое короткое из них в строке `sentence`:

```
string sentence = linel + line2 + line3;
```

Если несколько слов имеют длину, равную максимальной или минимальной, учтите их все.

6.9. Обрабатываем знаки препинания

После того как мы разбили каждую строку на слова, необходимо избавиться от знаков препинания. Пока из строки

```
magical but untamed. "Daddy, shush, there is no such thing,"
```

у нас получился такой набор слов:

```
magical
but
untamed.
"Daddy,
shush,
there
is
no
```

```
such
thing,"
```

Как нам теперь удалить ненужные знаки препинания? Для начала определим строку, содержащую все символы, которые мы хотим удалить:

```
string filt_elems( "\",.;:!?)(\\/" );
```

(Обратная косая черта указывает на то, что следующий за ней символ должен в данном контексте восприниматься буквально, а не как специальная величина. Так, \" обозначает символ двойной кавычки, а не конец строки, а \\ – символ обратной косой черты.)

Теперь можно применить функцию-член find_first_of() для поиска всех вхождений

```
while (( pos = word.find_first_of( filt_elems, pos ))
```

нежелательных символов:

```
            != string::npos )
```

Найденный символ удаляется с помощью функции-члена erase():

```
word.erase(pos,1);
```

Первый аргумент этой функции означает позицию подстроки, а второй – ее длину. Мы удаляем один символ, находящийся в позиции pos. Второй аргумент является необязательным; если его опустить, будут удалены все символы от pos до конца строки.

Вот полный текст функции filter_text(). Она имеет два параметра: указатель на

```
void
filter_text( vector<string> *words, string filter )
{
    vector<string>::iterator iter = words->begin();
    vector<string>::iterator iter_end = words->end();

    // Если filter не задан, зададим его сами
    if ( ! filter.size() )
        filter.insert( 0, "\"." );

    while ( iter != iter_end ) {
        string::size_type pos = 0;

        // удалим каждый найденный элемент
        while (( pos =  (*iter).find_first_of( filter, pos ))
                    != string::npos )
            (*iter).erase(pos,1);
        iter++;
    }
```

вектор строк, содержащий текст, и строку с символами, которые нужно убрать.

```
}
```

Почему мы не увеличиваем значение pos на каждой итерации? Что было бы, если бы мы написали:

```
while (( pos =  (*iter).find_first_of( filter, pos ))
           != string::npos )
{
     (*iter).erase(pos,1);
    ++ pos; // неправильно...
}
```

Возьмем строку

```
thing,"
```

На первой итерации pos получит значение 5 , т.е. позиции, в которой находится запятая. После удаления запятой строка примет вид

```
thing"
```

Теперь в 5-й позиции стоит двойная кавычка. Если мы увеличим значение pos, то пропустим этот символ.

```
string filt_elems( "\",.;:!?)(\\/" );
```

Так мы будем вызывать функцию filter_text():

```
filter_text( text_locations->first, filt_elems );
```

А вот часть распечатки, сделанной тестовой версией filter_text():

```
filter_text: untamed.
found! : pos: 7.
after: untamed

filter_text: "Daddy,
found! : pos: 0.
after: Daddy,
found! : pos: 5.
after: Daddy

filter_text: thing,"
found! : pos: 5.
after: thing"
found! : pos: 5.
after: thing

filter_text: "I
found! : pos: 0.
after: I

filter_text: Daddy,
found! : pos: 5.
after: Daddy

filter_text: there?"
found! : pos: 5.
after: there"
found! : pos: 5.
after: there
```

Упражнение 6.15

Напишите программу, которая удаляет все символы, кроме STL из строки:

```
"/.+(STL).$1/"
```

используя сначала erase(pos,count), а затем erase(iter,iter).

Упражнение 6.16

```
string sentence( "kind of" );
string s1 ( "whistle" )
```

Напишите программу, которая с помощью разных функций вставки из строк

```
string s2 ( "pixie" )
```

составит предложение

```
"A whistling-dixie kind of walk"
```

6.10. Приводим слова к стандартной форме

Одной из проблем при разработке текстовых поисковых систем является необходимость распознавать слова в различных словоформах, такие, как cry, cries и cried, baby и babies, и, что гораздо проще, написанные заглавными и строчными буквами, например home и Home. Первая задача, распознавание словоформ, слишком сложна, поэтому мы приведем здесь ее заведомо неполное решение. Сначала заменим все прописные буквы

```
void
strip_caps( vector<string,allocator> *words )
{
    vector<string,allocator>::iterator iter=words->begin() ;
    vector<string,allocator>::iterator iter_end=words->end() ;

    string caps( "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );

    while ( iter != iter_end ) {
        string::size_type pos = 0;
        while (( pos = (*iter).find_first_of( caps, pos ))
                    != string::npos )
            (*iter)[ pos ] = to1ower( (*iter)[pos] );
        ++iter;
    }
```

строчными:

```
}
```

Функция

```
to1ower( (*iter)[pos] );
```

входит в стандартную библиотеку C. Она заменяет прописную букву соответствующей ей строчной. Для использования `tolower()` необходимо включить заголовочный файл:

```
#include <ctype.h>
```

(В этом файле объявлены и другие функции, такие, как `isalpha()`, `isdigit()`, `ispunct()`, `isspace()`, `toupper()`. Полное описание этих функций см. [PLAUGER92]. Стандартная библиотека C++ включает класс `ctype`, который инкапсулирует всю функциональность стандартной библиотеки Си, а также набор функций, не являющихся членами, например `toupper()`, `tolower()` и т.д. Для их использования нужно включить заголовочный файл

```
#include <locale>
```

Однако наша реализация компилятора еще не поддерживала класс `ctype`, и нам пришлось использовать стандартную библиотеку Си.)

Проблема словоформ слишком сложна для того, чтобы пытаться решить ее в общем виде. Но даже самый примитивный вариант способен значительно улучшить работу нашей поисковой системы. Все, что мы сделаем в данном направлении, – удалим букву `'s'` на

```
void suffix_text( vector<string,allocator> *words )
{
    vector<string,allocator>::iterator
                  iter = words->begin(),
                  iter_end = words->end();

    while ( iter != iter_end ) {
        // оставим слова короче трех букв как есть
        if ( (*iter).size() <= 3 )
            { ++iter; continue; }
        if ( (*iter)[ (*iter).size()-1 ] == 's' )
            suffix_s( *iter );

        // здесь мы могли бы обработать суффиксы
        // ed, ing, 1y

        ++iter;
    }
```

концах слов:

```
}
```

Слова из трех и менее букв мы пропускаем. Это позволяет оставить без изменения, например, `has`, `its`, `is` и т.д., однако слова `tv` и `tvs` мы не сможем распознать как одинаковые.

```
string::size_type pos() = word.size()-3;
string ies( "ies" );
if ( ! word.compare( pos3, 3, ies )) {
    word.replace( pos3, 3, 1, 'y' );
    return;
```

Если слово кончается на `"ies"`, как `babies` и `cries`, необходимо заменить `"ies"` на `"y"`:

```
}
```

compare() возвращает 0, если две строки равны. Первый аргумент, pos3, обозначает начальную позицию, второй – длину сравниваемой подстроки (в нашем случае 3). Третий аргумент, ies, – строка-эталон. (На самом деле существует шесть вариантов функции compare(). Остальные мы покажем в следующем разделе.)

replace() заменяет подстроку набором символов. В данном случае мы заменяем подстроку "ies" длиной в 3 символа единичным символом 'y'. (Имеется десять перегруженных вариантов функции replace(). В следующем разделе мы коснемся остальных вариантов.)

Если слово заканчивается на "ses", как promises или purposes, нужно удалить

```
string ses( "ses" );
if ( ! word.compare( pos3, 3, ses )) {
    word.erase( pos3+l, 2 );
    return;
```

суффикс "es"[16]:

```
}
```

Если слово кончается на "ous", как oblivious, fulvous, cretaceous, или на "is", как genesis, mimesis, hepatitis, мы не будем изменять его. (Наша система несовершенна. Например, в слове kiwis надо убрать последнее 's'.) Пропустим и слова, оканчивающиеся на "ius" (genius) или на "ss" (hiss, lateness, less). Нам поможет

```
string::size_type spos = 0;
string::size_type pos3 = word.size()-3;

// "ous", "ss", "is", "ius"
string suffixes( "oussisius" );

if ( ! word.compare( pos3, 3, suffixes, spos, 3 ) ||     // ous
     ! word.compare( pos3, 3, suffixes, spos+6, 3 ) ||   // ius
     ! word.compare( pos3+l, 2, suffixes, spos+2, 2 ) || // ss
     ! word.compare( pos3+l, 2, suffixes, spos+4, 2 ) )  // is
```

вторая форма функции compare():

```
    return;


// удалим последнее 's'
```

В противном случае удалим последнее 's':

```
word.erase( pos3+2 );
```

16 Конечно, в английском языке существуют исключения из правил. Наш эвристический алгоритм превратит *crises* (множ. число от *crisis* – *прим. перев.*) в *cris*. Ошибочка!

Имена собственные, например Pythagoras, Brahms, Burne-Jones, не подпадают под общие правила. Этот случай мы оставим как упражнение для читателя, когда будем рассказывать об ассоциативных контейнерах.

Но прежде чем перейти к ним, рассмотрим оставшиеся строковые операции.

Упражнение 6.17

Наша программа не умеет обрабатывать суффиксы ed (surprised), ly (surprisingly) и ing (surprisingly). Реализуйте одну из функций для этого случая:

```
(a) suffix_ed()  (b) suffix_ly()    (c) suffix_ing()
```

6.11. Дополнительные операции со строками

Вторая форма функции-члена erase() принимает в качестве параметров два итератора, ограничивающих удаляемую подстроку. Например, превратим

```
string name( "AnnaLiviaPlurabelle" );
```

```
typedef string::size_type size_type;
size_type startPos = name.find( 'L' )
size_type endPos   = name.find_1ast_of( 'b' );

name.erase( name.begin()+startPos,
```

в строку "Annabelle":

```
            name.begin()+endPos );
```

Символ, на который указывает второй итератор, не входит в удаляемую подстроку.

Для третьей формы параметром является только один итератор; эта форма удаляет все символы, начиная с указанной позиции до конца строки. Например:

```
name.erase( name. begin()+4 );
```

оставляет строку "Anna".

Функция-член insert() позволяет вставить в заданную позицию строки другую строку или символ. Общая форма выглядит так:

```
string_object.insert( position, new_string );
```

position обозначает позицию, перед которой производится вставка. new_string может

```
string string_object( "Missisippi" );
string::size_type pos = string_object.find( "isi" );
```

быть объектом класса string, С-строкой или символом:

```
string_object.insert( pos+1, 's' );
```

```
string new_string ( "AnnaBelle Lee" );
string_object += ' '; // добавим пробел

// найдем начальную и конечную позицию в new_string
pos = new_string.find( 'B' );
string::size_type posEnd = new_string.find( ' ' );

string_object.insert(
    string_object.size(), // позиция вставки
    new_string, pos,      // начало подстроки в new_string
    posEnd                // конец подстроки new_string
)
```

Можно выделить для вставки подстроку из new_string:

string_object получает значение "Mississippi Belle". Если мы хотим вставить все символы new_string, начиная с pos, последний параметр нужно опустить.

```
string sl( "Mississippi" );
```

Пусть есть две строки:

```
string s2( "Annabelle" );
```

Как получить третью строку со значением "Miss Anna"?

```
string s3;

// скопируем первые 4 символа s1
```

Можно использовать функции-члены assign() и append():

```
s3.assign ( s1, 4 );


// добавим пробел
```

s3 теперь содержит значение "Miss".

```
s3 += ' ';


// добавим 4 первых символа s2
```

Теперь s3 содержит "Miss ".

```
s3.append(s2,4);
```

s3 получила значение "Miss Anna". То же самое можно сделать короче:

```
s3.assign(s1,4).append(' ').append(s2,4);
```

Другая форма функции-члена assign() имеет три параметра: второй обозначает позицию начала, а третий – длину. Позиции нумеруются с 0. Вот как, скажем, извлечь

```
string beauty;
// присвоим beauty значение "belle"
```

"belle" из "Annabelle":

```
beauty.assign( s2, 4, 5 );
```

```
// присвоим beauty значение "belle"
```

Вместо этих параметров мы можем использовать пару итераторов:

```
beauty.assign( s2, s2.begin()+4, s2.end() );
```

В следующем примере две строки содержат названия текущего проекта и проекта, находящегося в отложенном состоянии. Они должны периодически обмениваться

```
string current_project( "C++ Primer, 3rd Edition" );
```

значениями, поскольку работа идет то над одним, то над другим. Например:

```
string pending_project( "Fantasia 2000, Firebird segment" );
```

Функция-член swap() позволяет обменять значения двух строк с помощью вызова

```
current_project.swap( pending_project );
```

Для строки

```
string first_novel( "V" );
```

операция взятия индекса

```
char ch = first_novel[ 1 ];
```

возвратит неопределенное значение: длина строки first_novel равна 1, и единственное правильное значение индекса – 0. Такая операция взятия индекса не обеспечивает проверку правильности параметра, но мы всегда можем сделать это сами с помощью функции-члена size():

```
int
elem_count( const string &word, char elem )
{
    int occurs = 0;

    // не надо больше проверять ix
    for ( int ix=0; ix < word.size(); ++ix )
        if ( word[ ix ] == elem )
           ++occurs;
    return occurs;

}
```

```
void
mumble( const string &st, int index )
{
    // возможна ошибка
    char ch = st[ index ];

    // ...
```

Там, где это невозможно или нежелательно, например:

```
}
```

следует воспользоваться функцией `at()`, которая делает то же, что и операция взятия индекса, но с проверкой. Если индекс выходит за границу, возбуждается исключение

```
void
mumble( const string &st, int index )
{
    try {
        char ch = st.at( index );
        // ...
    }
    catch ( std::out_of_range ){...}
    // ...
```

`out_of_range`:

```
}
```

```
string cobol_program_crash( "abend" );
```

Строки можно сравнивать лексикографически. Например:

```
string cplus_program_crash( "abort" );
```

Строка `cobol_program_crash` лексикографически меньше, чем `cplus_program_crash`: сопоставление производится по первому отличающемуся символу, а буква `e` в латинском алфавите идет раньше, чем `o`. Операция сравнения выполняется функцией-членом `compare()`. Вызов

```
sl.compare( s2 );
```

возвращает одно из трех значений:

- если s1 больше, чем s2, то положительное;
- если s1 меньше, чем s2, то отрицательное;
- если s1 равно s2, то 0.

Например,

```
cobol_program_crash.compare( cplus_program_crash );
```

вернет отрицательное значение, а

```
cplus_program_crash.compare( cobol_program_crash );
```

положительное. Перегруженные операции сравнения (<, >, !=, ==, <=, >=) являются более компактной записью функции compare().

Шесть вариантов функции-члена compare() позволяют выделить сравниваемые подстроки в одном или обоих операндах. (Примеры вызовов приводились в предыдущем разделе.)

Функция-член replace() дает десять способов заменить одну подстроку на другую (их длины не обязаны совпадать). В двух основных формах replace() первые два аргумента задают заменяемую подстроку: в первом варианте в виде начальной позиции и длины, во втором – в виде пары итераторов на ее начало и конец. Вот пример первого

```
string sentence(
    "An ADT provides both interface and implementation." );

string::size_type position = sentence.find_1ast_of( 'A' );
string::size_type length = 3;

// заменяем ADT на Abstract Data Type
```

варианта:

```
sentence.repiace( position, length, "Abstract Data Type" );
```

position представляет собой начальную позицию, а length – длину заменяемой подстроки. Третий аргумент является подставляемой строкой. Его можно задать

```
string new_str( "Abstract Data Type" );
```

несколькими способами. Допустим, как объект string:

```
sentence.replace( position, length, new_str );
```

Следующий пример иллюстрирует выделение подстроки в new_str:

```
#include <string>
typedef string::size_type size_type;

// найдем позицию трех букв
size_type posA = new_str.find( 'A' );
size_type posD = new_str.find( 'D' );
size_type posT = new_str.find( 'T' );

// нашли: заменим T на "Type"
sentence.replace( position+2, 1, new_str, posT, 4 );

// нашли: заменим D на "Data "
sentence.replace( position+1, 1, new_str, posD, 5 );

// нашли: заменим A на "Abstract "
sentence.replace( position, 1, new_str, posA, 9 );
```

Еще один вариант позволяет заменить подстроку на один символ, повторенный заданное

```
string hmm( "Some celebrate Java as the successor to C++." );
string:: size_type position = hmm.find( 'J' );
// заменим Java на xxxx
```

количество раз:

```
hmm.repiace( position, 4, 'x', 4 );
```

В данном примере используется указатель на символьный массив и длина вставляемой

```
const char *lang = "EiffelAda95JavaModula3";
int index[] = { 0, 6, 11, 15, 22 };

string ahhem(
     "C++ is the language for today's power programmers." );
```

подстроки:

```
ahhem.replace(0, 3, lang+index[1], index[2]-index[1]);
```

```
string sentence(
    "An ADT provides both interface and implementation." );

// указывает на 'A' в ADT
string: iterator start = sentence. begin()+3;

// заменяем ADT на Abstract Data Type
```

А здесь мы используем пару итераторов:

```
sentence.repiace( start, start+3, "Abstract Data Type" );
```

Оставшиеся четыре варианта допускают задание заменяющей строки как объекта типа `string`, символа, повторяющегося N раз, пары итераторов и C-строки.

Вот и все, что мы хотели сказать об операциях со строками. Для более полной информации обращайтесь к определению стандарта C++ [ISO-C++97].

Упражнение 6.18

Напишите программу, которая с помощью функций-членов assign() и append() из

```
string quote1( "When lilacs last in the dooryard bloom'd" );
```

строк

```
string quote2( "The child "is father of the man" );
```

составит предложение

```
"The child is in the dooryard"
```

Упражнение 6.19

```
string generate_salutation( string generic1,
                            string lastname,
                            string generic2,
                            string::size_type pos,
```

Напишите функцию:

```
                            int length );
```

которая в строке

```
string generic1( "Dear Ms Daisy:" );
```

заменяет Daisy и Ms (миссис). Вместо Daisy подставляется параметр lastname, а вместо Ms подстрока

```
string generic2( "MrsMsMissPeople" );
```

длины length, начинающаяся с pos.

```
string lastName( "AnnaP" );
string greetings =
```

Например, вызов

```
    generate_salutation( generici, lastName, generic2, 5, 4 );
```

вернет строку:

```
Dear Miss AnnaP:
```

6.12. Строим отображение позиций слов

В этом разделе мы построим отображение (map), позволяющее для каждого уникального слова текста сохранить номера строк и колонок, в которых оно встречается. (В следующем разделе мы изучим ассоциативный контейнер set.) В общем случае контейнер set полезен, если мы хотим знать, содержится ли определенный элемент в некотором множестве, а map позволяет связать с каждым из них какую-либо величину.

В map хранятся пары ключ/значение. Ключ играет роль индекса для доступа к ассоциированному с ним значению. В нашей программе каждое уникальное слово текста будет служить ключом, а значением станет вектор, содержащий пары (номер строки, номер колонки). Для доступа применяется оператор взятия индекса. Например:

```
string query( "pickle" );
vector< location > *locat;

// возвращается location<vector>*, ассоциированный с "pickle"
locat = text_map[ query ];
```

Ключом здесь является строка, а значение имеет тип location<vector>*.

Для использования отображения необходимо включить соответствующий заголовочный файл:

```
#include <map>
```

Какие основные действия производятся над ассоциативными контейнерами? Их заполняют элементами или проверяют на наличие определенного элемента. В следующем подразделе мы покажем, как определить пару ключ/значение и как поместить такие пары в контейнер. Далее мы расскажем, как сформулировать запрос на поиск элемента и извлечь значение, если элемент существует.

6.12.1. Определение объекта map и заполнение его элементами

Чтобы определить объект класса map, мы должны указать, как минимум, типы ключа и значения. Например:

```
map<string,int> word_count;
```

Здесь задается объект word_count типа map, для которого ключом служит объект типа string, а ассоциированным с ним значением – объект типа int. Аналогично

```
class employee;
map<int,employee*> personnel;
```

определяет personnel как отображение ключа типа int (уникальный номер служащего) на указатель, адресующий объект класса employee.

```
typedef pair<short,short> location;
typedef vector<location> loc;
```

Для нашей поисковой системы полезно такое отображение:

```
map<string,loc*> text_map;
```

Поскольку имевшийся в нашем распоряжении компилятор не поддерживал аргументы по умолчанию для параметров шаблона, нам пришлось написать более развернутое

```
map<string,loc*,   // ключ, значение
    less<string>,  // оператор сравнения
    allocator>     // распределитель памяти по умолчанию
```

определение:

```
text_map;
```

По умолчанию сортировка ассоциативных контейнеров производится с помощью операции “меньше”. Однако можно указать и другой оператор сравнения (см. раздел 12.3 об объектах-функциях).

После того как отображение определено, необходимо заполнить его парами

```
#include <map>
#include <string>
map<string,int> word_count;

word_count[ string("Anna") ]  = 1;
word_count[ string("Danny") ] = 1;
word_count[ string("Beth") ]  = 1;
```

ключ/значение. Интуитивно хочется написать примерно так:

```
// и так далее ...
```

Когда мы пишем:

```
word_count[ string("Anna") ] = 1;
```

на самом деле происходит следующее:

1. Безымянный временный объект типа `string` инициализируется значением `"Anna"` и передается оператору взятия индекса, определенному в классе `map`.
2. Производится поиск элемента с ключом `"Anna"` в массиве `word_count`. Такого элемента нет.
3. В `word_count` вставляется новая пара ключ/значение. Ключом является, естественно, строка `"Anna"`. Значением – `0`, а не `1`.
4. После этого значению присваивается величина 1.

Если элемент отображения вставляется в отображение с помощью операции взятия индекса, то значением этого элемента становится значение по умолчанию для его типа данных. Для встроенных арифметических типов – 0.

Следовательно, если инициализация отображения производится оператором взятия индекса, то каждый элемент сначала получает значение по умолчанию, а затем ему явно присваивается нужное значение. Если элементы являются объектами класса, у которого инициализация по умолчанию и присваивание значения требуют больших затрат времени, программа будет работать правильно, но недостаточно эффективно.

```
// предпочтительный метод вставки одного элемента
word_count.insert(
    map<string,i nt>::
        value_type( string("Anna"), 1 )
);
```

Для вставки одного элемента предпочтительнее использовать следующий метод:

В контейнере map определен тип value_type для представления хранимых в нем пар

```
map< string,int >::
```

ключ/значение. Строки

```
    value_type( string("Anna"), 1 )
```

создают объект pair, который затем непосредственно вставляется в map. Для удобства чтения можно использовать typedef:

```
typedef map<string,int>::value_type valType;
```

Теперь операция вставки выглядит проще:

```
word_count.insert( valType( string("Anna"), 1 ));
```

Чтобы вставить элементы из некоторого диапазона, можно использовать метод insert(),

```
map< string, int > word_count;
// ... заполнить

map< string,int > word_count_two;

// скопируем все пары ключ/значение
```

принимающий в качестве параметров два итератора. Например:

```
word_count_two.insert(word_count.begin(),word_count.end());
```

Мы могли бы сделать то же самое, просто проинициализировав одно отображение другим:

```
// инициализируем копией всех пар ключ/значение
map< string, int > word_count_two( word_count );
```

Посмотрим, как можно построить отображение для хранения нашего текста. Функция separate_words(), описанная в разделе 6.8, создает два объекта: вектор строк, хранящий все слова текста, и вектор позиций, хранящий пары (номер строки, номер колонки) для каждого слова. Таким образом, первый объект дает нам множество значений ключей нашего отображения, а второй – множество ассоциированных с ними значений.

separate_words() возвращает эти два вектора как объект типа pair, содержащий указатели на них. Сделаем эту пару аргументом функции build_word_map(), в

```
// typedef для удобства чтения
typedef pair< short,short >  location;
typedef vector< location >   loc;
typedef vector< string >     text;
typedef pair< text*,loc* >   text_loc;

extern map< string, loc* >*
    build_word_map( const text_loc *text_locations );
```

результате которой будет получено соответствие между словами и позициями:

Сначала выделим память для пустого объекта map и получим из аргумента-пары

```
map<string,loc*> *word_map = new map< string, loc* >;
vector<string>   *text_words = text_locations->first;
```

указатели на векторы:

```
vector<location> *text_locs  = text_locations->second;
```

Теперь нам надо синхронно обойти оба вектора, учитывая два случая:

- слово встретилось впервые. Нужно поместить в map новую пару ключ/значение;
- слово встречается повторно. Нам нужно обновить вектор позиций, добавив дополнительную пару (номер строки, номер колонки).

Вот текст функции:

```
register int elem_cnt = text_words->size();
for ( int ix=0; ix < elem_cnt; ++ix )
{
    string textword = ( *text_words )[ ix ];

    // игнорируем слова короче трех букв
    // или присутствующие в списке стоп-слов
    if ( textword.size() < 3 ||
        exclusion_set.count( textword ))
           continue;

    // определяем, занесено ли слово в отображение
    // если count() возвращает 0 - нет: добавим его
    if ( ! word_map->count((*text_words)[-ix] ))
    {
        loc *ploc = new vector<location>;
        ploc->push_back( (*text_locs) [ix] );
        word_map->insert(value_type((*text_words)[ix],ploc));
    }
    else
    // добавим дополнительные координаты
    (*word_map)[(*text_words)[ix]]->
                        push_back((*text_locs)[ix]);
}
```

```
(*word_map)[(*text_words)[ix]]->
```

Синтаксически сложное выражение

```
                        push_back((*text_locs)[ix]);
```

```
// возьмем слово, которое надо обновить
string word = (*text_words) [ix];

// возьмем значение из вектора позиций
vector<location> *ploc = (*word_map) [ word ];

// возьмем позицию - пару координат
loc = (*text_locs)[ix];

// вставим новую позицию
```

будет проще понять, если мы разложим его на составляющие:

```
ploc->push_back(loc);
```

Выражение все еще остается сложным, так как наши векторы представлены указателями. Поэтому вместо употребления оператора взятия индекса:

```
string word = text_words[ix]; // ошибка
```

мы вынуждены сначала разыменовать указатель на вектор:

```
string word = (*text_words) [ix]; // правильно
```

В конце концов build_word_map() возвращает построенное отображение:

```
return word_map;
```

```
int main()
{
    // считываем файл и выделяем слова
    vector<string, allocator> *text_file = retrieve_text();
    text_loc *text_locations = separate_words( text_file );

    // обработаем слова
    // ...

    // построим отображение слов на векторы позиций
    map<string,loc*,less<string>,allocator>
         *text_map = build_word_map( text_locatons );

    // ...
```

Вот как выглядит вызов этой функции из main():

```
}
```

6.12.2. Поиск и извлечение элемента отображения

Оператор взятия индекса является простейшим способом извлечения элемента.

```
// map<string,int> word_count;
```

Например:

```
int count = word_count[ "wrinkles" ];
```

Однако этот способ работает так, как надо, только при условии, что запрашиваемый ключ действительно содержится в отображении. Иначе оператор взятия индекса поместит в отображение элемент с таким ключом. В данном случае в word_count занесется пара

```
string( "wrinkles" ), 0
```

Класс map предоставляет две операции для того, чтобы выяснить, содержится ли в нем определенное значение ключа.

- count(keyValue): функция-член count() возвращает количество элементов с данным ключом. (Для отображения оно равно только 0 или 1). Если count()

```
int count = 0;
if ( word_count.count( "wrinkles" ))
```

вернула 1, мы можем смело использовать индексацию:

```
    count = word_count[ "wrinkles" ];
```

- `find(keyValue)`: функция-член `find()` возвращает итератор, указывающий на

```
int count = 0;
map<string,int>::iterator it = word_count.find( "wrinkles" );
if ( it != word_count.end() )
```

элемент, если ключ найден, и итератор `end()` в противном случае. Например:

```
count = (*it).second;
```

Значением итератора является указатель на объект `pair`, в котором `first` содержит ключ, а `second` – значение. (Мы вернемся к этому в следующем подразделе.)

6.12.3. Навигация по элементам отображения

После того как мы построили отображение, хотелось бы распечатать его содержимое. Мы можем сделать это, используя итератор, начальное и конечное значение которого получают с помощью функций-членов `begin()` и `end()`. Вот текст функции

```
void
display_map_text( map<string,loc*> *text_map )
{
    typedef map<string,loc*> tmap;
    tmap::iterator iter = text_map->begin(),
    iter_end = text_map->end();

    while ( iter != iter_end )
    {
        cout << "word: " << (*iter).first << " (";
        int loc_cnt = 0;
        loc *text_locs = (*iter).second;
        loc::iterator liter = text_locs->begin(),
                      liter_end = text_locs->end();

        while (liter != liter_end ) {
            if ( loc_cnt )
                cout << ',';
            else ++loc_cnt;

            cout << '(' << (*liter).first
                 << ',' << (*liter).second << ')';

            ++liter;
        }

        cout << ")\n";
        ++iter;
    }
    cout << endl;
```

`display_map_text()`:

```
}
```

Если наше отображение не содержит элементов, данная функция не нужна. Проверить, пусто ли оно, можно с помощью функции-члена `size()`:

```
if ( text_map->size() )
    display_map_text( text_map );
```

Но более простым способом, без подсчета элементов, будет вызов функции-члена

```
if ( ! text_map->empty() )
```

`empty()`:

```
    display_map_text( text_map );
```

6.12.4. Словарь

Вот небольшая программа, иллюстрирующая построение отображения, поиск в нем и обход элементов. Здесь используются два отображения. Первое, необходимое для преобразования слов, содержит два элемента типа `string`. Ключом является слово, которое нуждается в специальной обработке, а значением – слово, заменяющее ключ. Для простоты мы задали пары ключ/значение непосредственно в тексте программы (вы можете модифицировать программу так, чтобы она читала их из стандартного ввода или из файла). Второе отображение используется для подсчета произведенных замен. Текст программы выглядит следующим образом:

```
#include <map>
#include <vector>
#include <iostream>
#include <string>

int main()
{
  map< string, string > trans_map;
  typedef map< string, string >::value_type valType;

  // первое упрощение:
  // жестко заданный словарь
  trans_map.insert( va1Type( "gratz", "grateful" ));
  trans_map.insert( va1Type( "'em",   "them"     ));
  trans_map.insert( va1Type( "cuz",   "because"  ));
  trans_map.insert( va1Type( "nah",   "no"       ));
  trans_map.insert( va1Type( "sez",   "says"     ));
  trans_map.insert( va1Type( "tanx",  "thanks"   ));
  trans_map.insert( va1Type( "wuz",   "was"      ));
  trans_map.insert( va1Type( "pos",   "suppose"  ));

  // напечатаем словарь
  map< string,string >::iterator it;

  cout << "Наш словарь подстановок: \n\n";
  for ( it = trans_map.begin();
          it != trans_map.end(); ++it )
      cout << "ключ: "   << (*it).first << "\t"
           << "значение: " << ("it).second << "\n";

  cout << "\n\n";

  // второе упрощение: жестко заданный текст
  string textarray[14]={ "nah", "I", "sez", "tanx",
              "cuz", "I", "wuz", "pos", "to", "not",
              "cuz", "I", "wuz", "gratz" };

  vector< string > text( textarray, textarray+14 );
  vector< string >::iterator iter;

  // напечатаем текст
  cout << "Исходный вектор строк:\n\n";
  int cnt = 1;
  for ( iter = text-begin(); iter != text.end();
                                      ++iter,++cnt )
      cout <<  *iter << ( cnt % 8 ? " " : "\n" );

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

  // map для сбора статистики
  map< string,int > stats;
  typedef map< string,int >::value_type statsValType;
  // здесь происходит реальная работа
  for ( iter=text.begin(); iter != text.end(); ++iter )
  if (( it = trans_map.find( *iter ))
            != trans_map.end() )
  {
      if ( stats.count( *iter ))
          stats [ *iter ] += 1;
      else stats.insert( statsVa1Type( *iter, 1 ));
      *iter = (*it).second;
  }

  // напечатаем преобразованный текст
  cout << "Преобразованный вектор строк:\n\n";
  cnt = 1;
  for ( iter = text.begin(); iter != text.end();
                                      ++iter, ++cnt )
      cout << *iter << ( cnt % 8 ? " " : "\n" );
  cout << "\n\n\n";

  // напечатаем статистику
  cout << "И напоследок статистика:\n\n";
  map<string,int,less<string>,allocator>::iterator siter;

  for (siter=stats.begin(); siter!=stats.end(); ++siter)
    cout << (*siter).first     << " "
```

```
}
```

Вот результат работы программы:

```
Наш словарь подстановок:

key: 'em      value: them
key: cuz      value: because
key: gratz    value: grateful
key: nah      value: no
key: pos      value: suppose
key: sez      value: says
key: tanx     value: thanks
key: wuz      value: was


Исходный вектор строк:
nah I sez tanx cuz I wuz pos
to not cuz I wuz gratz

Преобразованный вектор строк:
no I says thanks because I was suppose
to not because I was grateful


И напоследок статистика:

cuz было заменено 2 раз(а)
gratz было заменено 1 раз(а)
nah было заменено 1 раз(а)
pos было заменено 1 раз(а)
sez было заменено 1 раз(а)
tanx было заменено 1 раз(а)
wuz было заменено 2 раз(а)
```

6.12.5. Удаление элементов map

Существуют три формы функции-члена `erase()` для удаления элементов отображения. Для единственного элемента используется `erase()` с ключом или итератором в качестве аргумента, а для последовательности эта функция вызывается с двумя итераторами.

```
string removal_word;
cout << "введите удаляемое слово: ";
cin >> removal_word;

if ( text_map->erase( remova1_word ))
    cout << "ok: " << remova1_word << " удалено\n";
```

Например, мы могли бы позволить удалять элементы из `text_map` таким образом:

```
else cout << "увы: " << remova1_word << " не найдено!\n";
```

Альтернативой является проверка: действительно ли слово содержится в `text_map`?

```
map<string,loc*>::iterator where;
where = text_map.find( remova1_word );

if ( where == text_map->end() )
    cout << "увы: " << remova1_word << " не найдено!\n";
else {
    text_map->erase( where );
    cout << "ok: " << remova1_word << " удалено!\n";
}
```

В нашей реализации text_map с каждым словом сопоставляется множество позиций, что несколько усложняет их хранение и извлечение. Вместо этого можно было бы иметь по одной позиции на слово. Но контейнер map не допускает дублирующиеся ключи. Нам следовало бы воспользоваться классом multimap, который рассматривается в разделе 6.15.

Упражнение 6.20

Определите отображение, где ключом является фамилия, а значением – вектор с именами детей. Поместите туда как минимум шесть элементов. Реализуйте возможность делать запрос по фамилии, добавлять имена и распечатывать содержимое.

Упражнение 6.21

Измените программу из предыдущего упражнения так, чтобы вместе с именем ребенка записывалась дата его рождения: пусть вектор-значение хранит пары строк – имя и дата.

Упражнение 6.22

Приведите хотя бы три примера, в которых нужно использовать отображение. Напишите определение объекта map для каждого примера и укажите наиболее вероятный способ вставки и извлечения элементов.

6.13. Построение набора стоп-слов

Отображение состоит из пар ключ/значение. Множество (set), напротив, содержит неупорядоченную совокупность ключей. Например, бизнесмен может составить “черный список” bad_checks, содержащий имена лиц, в течение последних двух лет присылавших фальшивые чеки. Множество полезно тогда, когда нужно узнать, содержится ли определенное значение в списке. Скажем, наш бизнесмен, принимая чек от кого-либо, может проверить, есть ли его имя в bad_checks.

Для нашей поисковой системы мы построим набор стоп-слов – слов, имеющих семантически нейтральное значение (артикли, союзы, предлоги), таких, как *the*, *and*, *into*, *with*, *but* и т.д. (это улучшает качество системы, однако мы уже не сможем найти первое предложение из знаменитого монолога Гамлета: “To be or not to be?”). Прежде чем добавлять слово к word_map, проверим, не содержится ли оно в списке стоп-слов. Если содержится, проигнорируем его.

6.13.1. Определение объекта set и заполнение его элементами

Перед использованием класса set необходимо включить соответствующий заголовочный файл:

```
#include <set>
```

Вот определение нашего множества стоп-слов:

```
set<string> exclusion_set;
```

```
exclusion_set.insert( "the" );
```

Отдельные элементы могут добавляться туда с помощью операции `insert()`. Например:

```
exclusion_set.insert( "and" );
```

Передавая `insert()` пару итераторов, можно добавить целый диапазон элементов. Скажем, наша поисковая система позволяет указать файл со стоп-словами. Если такой

```
typedef set< string >::difference_type diff_type;
set< string > exclusion_set;

ifstream infile( "exclusion_set" );
if ( ! infile )
{
    static string default_excluded_words[25] = {
        "the","and","but","that","then","are","been",
        "can"."can't","cannot","could","did","for",
        "had","have","him","his","her","its","into",
        "were","which","when","with","would"
    };

    cerr << "предупреждение! невозможно открыть файл стоп-слов! -- "
         << "используется стандартный набор слов \n";

    copy( default_excluded_words, default_excluded_words+25,
        inserter( exclusion_set, exclusion_set.begin() ));
}
else {
    istream_iterator<string,diff_type> input_set(infile),eos;
    copy( input_set, eos, inserter( exclusion_set,
        exclusion_set.begin() ));
```

файл не задан, берется некоторый набор слов по умолчанию:

```
}
```

В этом фрагменте кода встречаются два элемента, которые мы до сих пор не рассматривали: тип `difference_type` и класс `inserter`. `difference_type` – это тип результата вычитания двух итераторов для нашего множества строк. Он передается в качестве одного из параметров шаблона `istream_iterator`.

`copy()` –один из обобщенных алгоритмов. (Мы рассмотрим их в главе 12 и в Приложении.) Первые два параметра – пара итераторов или указателей – задают диапазон. Третий параметр является либо итератором, либо указателем на начало контейнера, в который элементы копируются.

Проблема с этой функцией вызвана ограничением, вытекающим из ее реализации: количество копируемых элементов не может превосходить числа элементов в контейнере-адресате. Дело в том, что `copy()` не вставляет элементы, она только присваивает

каждому элементу новое значение. Однако ассоциативные контейнеры не позволяют явно задать размер. Чтобы скопировать элементы в наше множество, мы должны заставить copy() вставлять элементы. Именно для этого служит класс inserter (детально он рассматривается в разделе 12.4).

6.13.2. Поиск элемента

Две операции, позволяющие отыскать в наборе определенное значение, – это find() и count(). find() возвращает итератор, указывающий на найденный элемент, или значение, равное end(), если он отсутствует. count() возвращает 1 при наличии элемента и 0 в противном случае. Добавим проверку на

```
if ( exclusion_set.count( textword ))
     continue;
```

существование в функцию build_word_map():

```
// добавим отсутствующее слово
```

6.13.3. Навигация по множеству

Для проверки наших кодов реализуем небольшую функцию, выполняющую поиск по одному слову (поддержка языка запросов будет добавлена в главе 17). Если слово найдено, мы будем показывать каждую строку, в которой оно содержится. Слово может повторяться в строке, например:

```
tomorrow and tomorrow and tomorrow
```

однако такая строка будет представлена только один раз.

Одним из способов не учитывать повторное вхождение слова в строку является

```
// получим указатель на вектор позиций
loc ploc = (*text_map)[ query_text ];

// переберем все позиции
// вставим все номера строк в множество
set< short > occurrence_lines;
loc::iterator liter = ploc->begin(),
              liter_end = ploc->end();

while ( liter != liter_end ) {
    occurrence_lines.insert( occurrence_lines.end(),
          (*liter).first );
    ++liter;
```

использование множества, как показано в следующем фрагменте кода:

```
}
```

Контейнер set не допускает дублирования ключей. Поэтому можно гарантировать, что occurrence_lines не содержит повторений. Теперь нам достаточно перебрать данное

```
register int size = occurrence_lines.size();
cout << "\n" << query_text
     << " встречается " << size
     << " раз(а):")
     << "\n\n";

set< short >::iterator it=occurrence_lines.begin();
for ( ; it != occurrence_lines.end(); ++it ) {
    int line = -it;

    cout << "\t( строка "
         << line + 1 << " ) "
         << (*text_file)[line] << endl;
```

множество, чтобы показать все номера строк, где встретилось данное слово:

```
}
```

(Полная реализация query_text() представлена в следующем разделе.)

Класс set поддерживает операции size(), empty() и erase() точно таким же образом, как и класс map, описанный выше. Кроме того, обобщенные алгоритмы предоставляют набор специфических функций для множеств, например set_union() (объединение) и set_difference() (разность). (Они использованы при реализации языка запросов в главе 17.)

Упражнение 6.23

Добавьте в программу множество слов, в которых заключающее 's' не подчиняется общим правилам и не должно удаляться. Примерами таких слов могут быть Pythagoras, Brahms и Burne_Jones. Включите в функцию suffix_s() из раздела 6.10 проверку этого набора.

Упражнение 6.24

Определите вектор, содержащий названия книг, которые вы собираетесь прочесть в ближайшие шесть виртуальных месяцев, и множество, включающее названия уже прочитанных произведений. Напишите программу, которая выбирает для вас книгу из вектора при условии, что вы ее еще не прочитали. Выбранное название программа должна заносить в множество прочитанных. Однако вы могли отложить книгу; следовательно, нужно обеспечить возможность удалять ее название из множества прочитанных. По окончании шести виртуальных месяцев распечатайте список прочитанного и непрочитанного.

6.14. Окончательная программа

Ниже представлен полный текст программы, разработанной в этой главе, с двумя модификациями: мы инкапсулировали все структуры данных и функции в класс TextQuery (в последующих главах мы обсудим подобное использование классов), кроме того, текст был изменен, так как наш компилятор поддерживал стандарт C++ не полностью.

Например, библиотека `iostream` не соответствовала текущему стандарту. Шаблоны не поддерживали значения аргументов по умолчанию. Возможно, вам придется изменить кое-что в этой программе, чтобы она компилировалась в вашей системе.

```
// стандартные заголовочные файлы C++
#include <algorithm>
#include <string>
#include <vector>
#include <utility>
#include <map>
#include <set>

// заголовочный файл iostream, не отвечающий стандарту
#include <fstream.h>

// заголовочные файлы C
#include <stddef.h>
#include <ctype.h>

// typedef для удобства чтения
typedef pair<short,short>           location;
typedef vector<location,allocator>  loc;
typedef vector<string,allocator>    text;
typedef pair<text*,loc*>            text_loc;

class TextQuery {
public:
    TextQuery() { memset( this, 0, sizeof( TextQuery )); }

    static void
      filter_elements( string felems ) { filt_elems = felems; }

    void query_text();
    void display_map_text();
    void display_text_locations();
    void doit() {
        retrieve_text();
        separate_words();
        filter_text();
        suffix_text();
        strip_caps();
        build_word_map();
    }

private:
    void retrieve_text();
    void separate_words():
    void filter_text();
    void strip_caps();
    void suffix_textQ;
    void suffix_s( string& );
    void build_word_map();

private:
    vector<string,allocator>      *lines_of_text;
    text_loc                      *text_locations;
    map< string,loc*,
        less<string>,allocator>   *word_map;
    static string                 filt_elems;
};

string TextQuery::filt_elems( "\", •;: !?)(\V" );

int main()
{
    TextQuery tq;
    tq.doit();
    tq.query_text();
    tq.display_map_text();
}

void
TextQuery::
retrieve_text()
{
    string file_name;

    cout << "please enter file name: ";
    cin >> file_name;

    ifstream infile( file name.c str(), ios::in );
```

```
}
```

Упражнение 6.25

Объясните, почему нам потребовался специальный класс `inserter` для заполнения

```
set<string> exclusion_set;
ifstream    infile( "exclusion_set" );

copy( default_excluded_words, default_excluded_words+25,
      inserter(exclusion_set, exclusion_set.begin() ));
```

набора стоп-слов (это упоминается в разделе 6.13.1, а детально рассматривается в 12.4.1).

Упражнение 6.26

Первоначальная реализация поисковой системы отражает процедурный подход: набор глобальных функций оперирует набором независимых структур данных. Окончательный вариант представляет собой альтернативный подход, когда мы инкапсулируем функции и данные в класс `TextQuery`. Сравните оба способа. Каковы недостатки и преимущества каждого?

Упражнение 6.27

В данной версии программы имя файла с текстом вводится по запросу. Более удобно было бы задавать его как параметр командной строки; в главе 7 мы покажем, как это делается. Какие еще параметры командной строки желательно реализовать?

6.15. Контейнеры multimap и multiset

Контейнеры `map` и `set` не допускают повторяющихся значений ключей, а `multimap` (мультиотображение) и `multiset` (мультимножество) позволяют сохранять ключи с дублирующимися значениями. Например, в телефонном справочнике может понадобиться отдельный список номеров для каждого абонента. В перечне книг одного автора может быть несколько названий, а в нашей программе с одним словом текста сопоставляется несколько позиций. Для использования `multimap` и `multiset` нужно

```
#include <map>
multimap< key_type, value_type > multimapName;

// ключ - string, значение - list< string >
multimap< string, list< string > > synonyms;

#include <set>
```

включить соответствующий заголовочный файл – `map` или `set`:

```
multiset< type > multisetName;
```

Для прохода по мультиотображению или мультимножеству можно воспользоваться комбинацией итератора, который возвращает `find()` (он указывает на первый найденный элемент), и значения, которое возвращает `count()`. (Это работает, поскольку в данных контейнерах элементы с одинаковыми ключами обязательно являются соседними). Например:

```
#include <map>
#include <string>

void code_fragment()
{
    multimap< string, string > authors;
    string search_item(  "Alain de Botton" );
    // ...
    int number = authors.count( search_item );
    mu1timap< string,string >::iterator iter;

    iter = authors.find( search_item );
    for ( int cnt = 0; cnt < number; ++cnt, ++-iter )
        do_something( *iter );
    // ...
}
```

Более элегантный способ перебрать все значения с одинаковыми ключами использует специальную функцию-член `equal_range()`, которая возвращает пару итераторов. Один из них указывает на первое найденное значение, а второй – на следующее за последним найденным. Если последний из найденных элементов является последним в контейнере, второй итератор содержит величину, равную `end()`:

```
#include <map>
#include <string>
#include <utility>

void code_fragment()
{
    multimap< string, string > authors;
    // ...
    string search_item( "Haruki Murakami" );

    while ( cin && cin >> search_item )
        switch ( authors.count( search_item ))
        {
          // не найдено
          case 0:
            break;

         // найден 1, обычный find()
         case 1: {
           multimap< string, string >: iterator iter;
           iter = authors.find( search_item );
           // обработка элемента ...
           break;
         }
         // найдено несколько ...
         default:
         {
            typedef multimap<string,string>::iterator iterator;
            pair< iterator, iterator > pos;

            // pos.first - адрес 1-го найденного
            // pos.second - адрес 1-го отличного
            // от найденного
            pos = authors.equa1_range( search_item );
            for (; pos.first != pos.second; pos.first++ )
                // обработка элемента ...
         }
       }
}
```

Вставка и удаление элементов в `multimap` и `multiset` ничем не отличаются от аналогичных операций с контейнерами `map` и `set`. Функция `equal_range()` доставляет

```
#include <multimap>
#include <string>

typedef multimap< string, string >::iterator iterator;
pair< iterator, iterator > pos;
string search_item( "Kazuo Ishiguro" );

// authors - multimap<string, string>
// эквивалентно
// authors.erase( search_item );
pos = authors.equa1_range( search_item );
```

итераторную пару, задающую диапазон удаляемых элементов:

```
authors.erase( pos.first, pos.second );
```

При каждом вызове функции-члена `insert()` добавляется новый элемент, даже если в

```
typedef multimap<string,string>::value_type valType;
multimap<string,string> authors;

// первый элемент с ключом Barth
authors.insert( valType (
    string( "Barth, John" ),
    string( "Sot-Weed Factor" )));

// второй элемент с ключом Barth
authors.insert( va1Type(
    string( "Barth, John" ),
```

контейнере уже был элемент с таким же ключом. Например:

```
    string( "Lost in the Funhouse" )));
```

Контейнер `multimap` не поддерживает операцию взятия индекса. Поэтому следующее выражение ошибочно:

```
authors[ "Barth, John" ]; // ошибка: multimap
```

Упражнение 6.28

Перепишите программу текстового поиска из раздела 6.14 с использованием `multimap` для хранения позиций слов. Каковы производительность и дизайн в обоих случаях? Какое решение вам больше нравится? Почему?

6.16. Стек

В разделе 4.5 операции инкремента и декремента были проиллюстрированы на примере реализации абстракции стека. В общем случае стек является очень полезным механизмом для сохранения текущего состояния, если в разные моменты выполнения программы одновременно существует несколько состояний, вложенных друг в друга. Поскольку стек – это важная абстракция данных, в стандартной библиотеке C++ предусмотрен класс `stack`, для использования которого нужно включить заголовочный файл:

```
#include <stack>
```

В стандартной библиотеке стек реализован несколько иначе, чем у нас. Разница состоит в том, что доступ к элементу с вершины стека и удаление его осуществляются двумя функциями – `top()` и `pop()`. Полный набор операций со стеком приведен в таблице 6.5.

Таблица 6.5. Операции со стеком

Операция	Действие
`empty()`	Возвращает `true`, если стек пуст, и `false` в противном случае
`size()`	Возвращает количество элементов в стеке
`pop()`	Удаляет элемент с вершины стека, но не возвращает его значения
`top()`	Возвращает значение элемента с вершины

	стека, но не удаляет его
`push(item)`	Помещает новый элемент в стек

```
#include <stack>
#include <iostream>
int main()
{
const int ia_size = 10;
int ia[ia_size ]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
// заполним стек
int ix = 0;
stack< int > intStack;
for ( ; ix < ia_size; ++ix )
    intStack.push( ia[ ix ] );

int error_cnt = 0;
if ( intStack.size() != ia_size ) {
    cerr << "Ошибка! неверный размер IntStack: "
         << intStack.size()
         << "\t ожидается: " << ia_size << endl,
    ++error_cnt;
}

int value;
while ( intStack.empty() == false )
{
    // считаем элемент с вершины
    value = intStack.top();
    if ( value != --ix ) {
        cerr << "Ошибка! ожидается " << ix
             << " получено " << value << endl;
        ++error_cnt;
    }

    // удалим элемент
    intStack.pop();
}

cout << "В результате запуска программы получено "
<< error_cnt << " ошибок" << endl;
```

В нашей программе приводятся примеры использования этих операций:

```
}
```

Объявление

```
stack< int > intStack;
```

определяет `intStack` как пустой стек, предназначенный для хранения элементов типа `int`. Стек является надстройкой над некоторым контейнерным типом, поскольку реализуется с помощью того или иного контейнера. По умолчанию это `deque`, поскольку именно эта структура обеспечивает эффективную вставку и удаление первого элемента, а `vector` эти операции не поддерживает. Однако мы можем явно указать другой тип контейнера, задав его как второй параметр:

```
stack< int, list<int> > intStack;
```

Элементы, добавляемые в стек, копируются в реализующий его контейнер. Это может приводить к потере эффективности для больших или сложных объектов, особенно если мы только читаем элементы. В таком случае удобнее определить стек указателей на

```
#include <stack>
class NurbSurface { /* mumble */ };
```

объекты. Например:

```
stack< NurbSurface* > surf_Stack;
```

К двум стекам одного типа можно применять операции сравнения: равенство, неравенство, меньше, больше, меньше или равно, больше или равно, если они определены над элементами стека. Элементы сопоставляются попарно. Первая пара несовпадающих элементов определяет результат операции сравнения в целом.

Стек будет использован в нашей программе текстового поиска в разделе 17.7 для поддержки сложных запросов типа

```
Civil && ( War || Rights )
```

6.17. Очередь и очередь с приоритетами

Абстракция очереди реализует метод доступа FIFO (first in, first out – “первым вошел, первым вышел”): объекты добавляются в конец очереди, а извлекаются из начала. Стандартная библиотека предоставляет две разновидности этого метода: очередь FIFO, или простая очередь, и очередь с приоритетами, которая позволяет сопоставлять элементы с их приоритетами. Текущий элемент помещается не в конец такой очереди, а перед элементами с более низким приоритетом. Программист, определяющий такую структуру, задает способ вычисления приоритетов. В реальной жизни подобное можно увидеть, скажем, при регистрации багажа в аэропорту. Как правило, пассажиры, чей рейс через 15 минут, передвигаются в начало очереди, чтобы не опоздать на самолет. Примером из практики программирования служит планировщик операционной системы, определяющий последовательность выполнения процессов.

Для использования `queue` и `priority_queue` необходимо включить заголовочный файл:

```
#include <queue>
```

Полный набор операций с контейнерами `queue` и `priority_queue` приведен в таблице 6.6.

Таблица 6.6. Операции с `queue` и `priority_queue`

Операция	Действие
`empty()`	Возвращает `true`, если очередь пуста, и `false` в противном случае

`size()`	Возвращает количество элементов в очереди
`pop()`	Удаляет первый элемент очереди, но не возвращает его значения. Для очереди с приоритетом первым является элемент с наивысшим приоритетом
`front()`	Возвращает значение первого элемента очереди, но не удаляет его. Применимо только к простой очереди
`back()`	Возвращает значение последнего элемента очереди, но не удаляет его. Применимо только к простой очереди
`top()`	Возвращает значение элемента с наивысшим приоритетом, но не удаляет его. Применимо только к очереди с приоритетом
`push(item)`	Помещает новый элемент в конец очереди. Для очереди с приоритетом позиция элемента определяется его приоритетом.

Элементы `priority_queue` отсортированы в порядке убывания приоритетов. По умолчанию упорядочение основывается на операции “меньше”, определенной над парами элементов. Конечно, можно явно задать указатель на функцию или объект-функцию, которая будет использоваться для сортировки. (В разделе 12.3 можно найти более подробное объяснение и иллюстрации использования такой очереди.)

6.18. Вернемся в классу iStack

У класса `iStack`, разработанного нами в разделе 4.15, два недостатка:

- он поддерживает только тип `int`. Мы хотим обеспечить поддержку любых типов. Это можно сделать, преобразовав наш класс в шаблон класса `Stack`;
- он имеет фиксированную длину. Это неудобно в двух отношениях: заполненный стек становится бесполезным, а в попытке избежать этого мы окажемся перед необходимостью отвести ему изначально слишком много памяти. Разумным выходом будет разрешить динамический рост стека. Это можно сделать, пользуясь тем, что лежащий в основе стека вектор способен динамически расти.

Напомним определение нашего класса `iStack`:

```
#include <vector>

class iStack {
public:
    iStack( int capacity )
              : _stack( capacity ), _top( 0 ) {};

    bool pop( int &value );
    bool push( int value );

    bool full();
    bool empty();
    void display();

    int size();

private:
    int _top;
    vector< int > _stack;

};
```

Сначала реализуем динамическое выделение памяти. Тогда вместо использования индекса при вставке и удалении элемента нам нужно будет применять соответствующие функции-члены. Член `_top` больше не нужен: функции `push_back()` и `pop_back()` автоматически работают в конце массива. Вот модифицированный текст функций `pop()`

```
bool iStack::pop( int &top_value )
{
    if ( empty() )
        return false;
    top_value = _stack.back(); _stack.pop_back();
    return true;
}

bool iStack::push( int value )
{
    if ( full() )
        return false;
    _stack.push_back( value );
    return true;
}
```

и `push()`:

Функции-члены `empty()`, `size()` и `full()` также нуждаются в изменении: в этой версии

```
inline bool iStack::empty(){ return _stack.empty(); }
inline bool iStack::size() { return _stack.size(); }
inline bool iStack::full() {
    return _stack.max_size() == _stack.size();   }
```

они теснее связаны с лежащим в основе стека вектором.

Надо немного изменить функцию-член `display()`, чтобы `_top` больше не фигурировал в качестве граничного условия цикла.

```
void iStack::display()
{
    cout << "( " << size() << " )( bot: ";
    for ( int ix=0; ix < size(); ++ix )
        cout << _stack[ ix ] << " ";
    cout << " stop )\n";
}
```

Наиболее существенным изменениям подвергнется конструктор iStack. Никаких действий от него теперь не требуется. Можно было бы определить пустой конструктор:

```
inline iStack::iStack() {}
```

Однако это не совсем приемлемо для пользователей нашего класса. До сих пор мы строго сохраняли интерфейс класса iStack, и если мы хотим сохранить его до конца, необходимо оставить для конструктора один необязательный параметр. Вот как будет

```
class iStack {
public:
    iStack( int capacity = 0 );
    // ...
};
```

выглядеть объявление конструктора с таким параметром типа int:

```
inline iStack::iStack( int capacity )
{
    if ( capacity )
        _stack.reserve( capacity );
}
```

Что делать с аргументом, если он задан? Используем его для указания емкости вектора:

Превращение класса в шаблон еще проще, в частности потому, что лежащий в основе

```
#include <vector>

template <class elemType>
class Stack {
public:
    Stack( int capacity=0 );
    bool pop( elemType &value );
    bool push( elemType value );

    bool full();
    bool empty();
    void display();

    int size();
private:
    vector< elemType > _stack;
```

вектор сам является шаблоном. Вот модифицированное объявление:

```
};
```

Для обеспечения совместимости с программами, использующими наш прежний класс `iStack`, определим следующий `typedef`:

```
typedef Stack<int> iStack;
```

Модификацию операторов класса мы оставим читателю для упражнения.

Упражнение 6.29

Модифицируйте функцию `peek()` (упражнение 4.23 из раздела 4.15) для шаблона класса `Stack`.

Упражнение 6.30

Модифицируйте операторы для шаблона класса `Stack`. Запустите тестовую программу из раздела 4.15 для новой реализации

Упражнение 6.31

По аналогии с классом `List` из раздела 5.11.1 инкапсулируйте наш шаблон класса `Stack` в пространство имен `Primer_Third_Edition`

Часть III

Процедурно-ориентированное программирование

В части II были представлены базовые компоненты языка C++: встроенные типы данных (`int` и `double`), типы классов (`string` и `vector`) и операции, которые можно совершать над данными. В части III мы увидим, как из этих компонентов строятся функции, служащие для реализации алгоритмов.

В каждой программе на C++ должна присутствовать функция `main()`, которая получает управление при запуске программы. Все остальные функции, необходимые для решения задачи, вызываются из `main()`. Они обмениваются информацией при помощи *параметров*, которые получают при вызове, и возвращаемых значений. В главе 7 представлен соответствующие механизмы C++.

Функции используются для того, чтобы организовать программу в виде совокупности небольших и не зависящих друг от друга частей. Она инкапсулирует алгоритм или набор алгоритмов, применяемых к некоторому набору данных. Объекты и типы можно определить так, что они будут использоваться в течение всего времени работы программы. Однако, если некоторые объекты или типы применяются только в части программы, предпочтительнее ограничить область их использования именно этой частью и объявить внутри той функции, где они нужны. Понятие *видимости* предоставляет в распоряжение программиста механизм, позволяющий ограничивать область применения объектов. Различные области видимости, поддерживаемые языком C++, мы рассмотрим в главе 8.

Для облегчения использования функций C++ предлагает множество средств, рассматриваемых нами в части III. Первым из них является перегрузка. Функции, которые выполняют семантически одну и ту же операцию, но работают с разными типами данных и потому имеют несколько отличающиеся реализации, могут иметь общее имя. Например, все функции для печати значений разных типов, таких, как `int`, `string` и т.д., называются `print()`. Поскольку программисту не приходится запоминать много

разных имен для одной и той же операции, пользоваться ими становится проще. Компилятор сам подставляет нужное в зависимости от типов фактических аргументов. В главе 9 объясняется, как объявлять и использовать перегруженные функции и как компилятор выбирает подходящую из набора перегруженных.

Вторым средством, облегчающим использование функций, является механизм шаблонов. Шаблон – это обобщенное определение, которое используется для *конкретизации* – автоматической генерации потенциально бесконечного множества функций, различающихся только типами входных данных, но не действиями над ними. Этот механизм описывается в главе 10.

Функции обмениваются информацией с помощью значений, которые они получают при вызове (параметров), и значений, которые они возвращают. Однако этот механизм может оказаться недостаточным при возникновении непредвиденной ситуации в работе программы. Такие ситуации называются *исключениями*, и, поскольку они требуют немедленной реакции, необходимо иметь возможность послать сообщение вызывающей программе. Язык C++ предлагает механизм обработки исключений, который позволяет функциям общаться между собой в таких условиях. Этот механизм рассматривается в главе 11.

Наконец, стандартная библиотека предоставляет нам обширный набор часто используемых функций – обобщенных алгоритмов. В главе 12 описываются эти алгоритмы и способы их использования с контейнерными типами из главы 6 и со встроенными массивами.

7. Функции

Мы рассмотрели, как объявлять переменные (глава 3), как писать выражения (глава 4) и инструкции (глава 5). Здесь мы покажем, как группировать эти компоненты в определения функций, чтобы облегчить их многократное использование внутри программы. Мы увидим, как объявлять и определять функции и как вызывать их, рассмотрим различные виды передаваемых параметров и обсудим особенности использования каждого вида. Мы расскажем также о различных видах значений, которые может вернуть функция. Будут представлены четыре специальных случая применения функций: встроенные (inline), рекурсивные, написанные на других языках и объявленные директивами связывания, а также функция `main()`. В завершение главы мы разберем более сложное понятие – указатель на функцию.

7.1. Введение

Функцию можно рассматривать как операцию, определенную пользователем. В общем случае она задается своим именем. Операнды функции, или *формальные параметры*, задаются в *списке параметров*, через запятую. Такой список заключается в круглые скобки. Результатом функции может быть значение, которое называют *возвращаемым*. Об отсутствии возвращаемого значения сообщают ключевым словом `void`. Действия, которые производит функция, составляют ее *тело*; оно заключено в фигурные скобки. Тип возвращаемого значения, ее имя, список параметров и тело составляют *определение функции*. Вот несколько примеров:

```
inline int abs( int obj )
{
    // возвращает абсолютное значение iobj
    return( iobj < 0 ? -iobj : iobj );
}
inline int min( int p1, int p2 )
{
    // возвращает меньшую из двух величин
    return( pi < p2 ? pi :    p2 );
}

int gcd( int vl, int v2 )
{
    // возвращает наибольший общий делитель
    while ( v2 )
    {
        int temp = v2;
        v2 = vl % v2;
        vl = temp;
    }
    return vl;
}
```

Выполнение функции происходит тогда, когда в тексте программы встречается оператор вызова. Если функция принимает параметры, при ее вызове должны быть указаны *фактические параметры*, аргументы. Их перечисляют внутри скобок, через запятую. В следующем примере `main()` дважды вызывает `abs()` и по одному разу `min()` и `gcd()`.

```
#include <iostream>

int main()
{
    // прочитать значения из стандартного ввода
    cout << "Введите первое значение: ";
    int i;
    cin >> i;
    if ( !cin ) {
        cerr << "!? Ошибка ввода - аварийный выход!\n";
        return -1;
    }

    cout << "Введите второе значение: ";
    int j;
    cin >> j;
    if ( !cin ) {
        cerr << "!? Ошибка ввода - аварийный выход!\n";
        return -2;
    }

    cout << "\nmin: " << min( i, j ) << endl;
    i = abs( i );
    j = abs( j );
    cout << "НОД: " << gcd( i, j ) << endl;
    return 0;
}
```

Функция `main()` определяется в файле `main.C`.

Вызов функции может обрабатываться двумя разными способами. Если она объявлена *встроенной* (inline), то компилятор подставляет в точку вызова ее тело. Во всех остальных случаях происходит нормальный вызов, который приводит к передаче управления ей, а активный в этот момент процесс на время приостанавливается. По завершении работы выполнение программы продолжается с точки, непосредственно следующей за точкой вызова. Работа функции завершается выполнением последней инструкции ее тела или специальной инструкции `return`.

Функция должна быть объявлена до момента ее вызова, попытка использовать необъявленное имя приводит к ошибке компиляции. Определение функции может служить ее объявлением, но ему разрешено появиться в программе только один раз. Поэтому обычно его помещают в отдельный исходный файл. Иногда в одном файле находятся определения нескольких функций, логически связанных друг с другом. Чтобы использовать их в другом исходном файле, необходим механизм, позволяющий объявить ее, не определяя.

Объявление функции состоит из типа возвращаемого значения, имени и списка параметров. Вместе эти три элемента составляют *прототип*. Объявление может появиться в файле несколько раз.

В нашем примере файл main.C не содержит определений `abs()`, `min()` и `gcd()`, поэтому вызов любой из них приводит к ошибке компиляции. Чтобы компиляция была успешной,

```
int abs( int );
int min( int, int );
```

их необязательно определять, достаточно только объявить:

```
int gcd( int, int );
```

(В таком объявлении можно не указывать имя параметра, ограничиваясь названием типа.)

Объявления (а равно определения встроенных функций[17]) лучше всего помещать в заголовочные файлы, которые могут включаться всюду, где необходимо вызвать функцию. Таким образом, все файлы используют одно общее объявление. Если его необходимо модифицировать, изменения будут локализованы. Вот так выглядит

```
// определение функции находится в файле gcd.C
int gcd( int, int );

inline int abs(int i) {
    return( i<0 ? -i : i );
}
inline int min(int vl.int v2) {
    return( vl<v2 ? vl : v2 );
```

заголовочный файл для нашего примера. Назовем его `localMath.h`:

```
}
```

---

17 Таким образом, как мы видим, определения встроенных функций могут встретиться в программе несколько раз! – *Прим. ред*.

В объявлении функции описывается ее *интерфейс*. Он содержит все данные о том, какую информацию должна получать функция (список параметров) и какую информацию она возвращает. Для пользователей важны только эти данные, поскольку лишь они фигурируют в точке вызова. Интерфейс помещается в заголовочный файл, как мы поступили с функциями `min()`, `abs()` и `gcd()`.

При выполнении наша программа `main.C`, получив от пользователя значения:

```
Введите первое значение: 15
Введите второе значение: 123
```

выдаст следующий результат:

```
mm: 15
НОД: 3
```

7.2. Прототип функции

Прототип функции описывает ее интерфейс и состоит из типа возвращаемого функцией значения, имени и списка параметров. В данном разделе мы детально рассмотрим эти характеристики.

7.2.1. Тип возвращаемого функцией значения

Тип возвращаемого функцией значения бывает встроенным, как `int` или `double`, составным, как `int&` или `double*`, или определенным пользователем – перечислением или классом. Можно также использовать специальное ключевое слово `void`, которое

```
#include <string>
#include <vector> class Date { /* определение */ };

bool look_up( int *, int );
double calc( double );
int count( const string &, char );
Date& calendar( const char );
```

говорит о том, что функция не возвращает никакого значения:

```
void sum( vector<int>&, int );
```

Однако функция или встроенный массив не могут быть типом возвращаемого значения.

```
// массив не может быть типом возвращаемого значения
```

Следующий пример ошибочен:

```
int[10] foo_bar();
```

Но можно вернуть указатель на первый элемент массива:

```
// правильно: указатель на первый элемент массива
int *foo_bar();
```

(Размер массива должен быть известен вызывающей программе.)

```
// правильно: возвращается список символов
```

Функция может возвращать типы классов, в частности контейнеры. Например:

```
list<char> foo_bar();
```

(Этот подход не очень эффективен. Обсуждение типа возвращаемого значения см. в разделе 7.4.)

Тип возвращаемого функцией значения должен быть явно указан. Приведенный ниже

```
// ошибка: пропущен тип возвращаемого значения
```

код вызывает ошибку компиляции:

```
const is_equa1( vector<int> vl, vector<int> v2 );
```

В предыдущих версиях C++ в подобных случаях считалось, что функция возвращает значение типа `int`. Стандарт C++ отменил это соглашение. Правильное объявление

```
// правильно: тип возвращаемого значения указан
```

`is_equal()` выглядит так:

```
const bool is_equa1( vector<int> vl, vector<int> v2 );
```

7.2.2. Список параметров функции

Список параметров не может быть опущен. Функция, которая не требует параметров, должна иметь пустой список либо список, состоящий из одного ключевого слова `void`.

```
int fork();
```

Например, следующие объявления эквивалентны:

```
int fork( void );
```

Такой список состоит из названий типов, разделенных запятыми. После имени типа может находиться имя параметра, хотя это и необязательно. В списке параметров не разрешается использовать сокращенную запись, соотнося одно имя типа с несколькими

```
int manip( int vl, v2 );     // ошибка
```

параметрами:

```
int manip( int vl, int v2 ); // правильно
```

Имена параметров не могут повторяться. Имена, фигурирующие в определении функции, можно и даже нужно использовать в ее теле. В объявлении же функции они не обязательны и служат средством документирования ее интерфейса. Например:

```
void print( int *array, int size );
```

Имена параметров в объявлении и в определении одной и той же функции не обязаны совпадать. Однако употребление разных имен может запутать пользователя.

C++ допускает сосуществование двух или более функций, имеющих одно и то же имя, но разные списки параметров. Такие функции называются *перегруженными*. О списке параметров в этом случае говорят как о *сигнатуре* функции, поскольку именно он используется различения разных версий одноименных функций. Имя и сигнатура однозначно идентифицируют версию. (Перегруженные функции подробно обсуждаются в главе 9.)

7.2.3. Проверка типов формальных параметров

Функция `gcd()` объявлена следующим образом:

```
int gcd( int, int );
```

Объявление говорит о том, что имеется два параметра типа `int`. Список формальных параметров предоставляет компилятору информацию, с помощью которой тот может проверить типы передаваемых функции фактических аргументов.

Что будет, если попытаться вызвать функцию `gcd()` с аргументами типа `char*`?

```
gcd( "hello", "world" );
```

А если передать этой функции не два аргумента, а только один? Или больше двух? Что случится, если потеряется запятая между числами `24` и `312`?

```
gcd( 24312 );
```

Единственное разумное поведение компилятора – сообщение об ошибке, поскольку попытка выполнить такую программу чревата весьма серьезными последствиями. C++ действительно не пропустит подобные вызовы. Текст сообщения будет выглядеть примерно так:

```
// gcd( "hello", "world" )
error: invalid argument types ( const char *, const char * ) --
       expecting ( int, int )
ошибка: неверные типы аргументов ( const char *, const char * ) --
       ожидается ( int, int )

// gcd( 24312 )
error: missing value for second argument
ошибка: пропущено значение второго аргумента
```

А если вызвать эту функцию с аргументами типа `double`? Должен ли этот вызов расцениваться как ошибочный?

```
gcd( 3.14, 6.29 );
```

Как было сказано в разделе 4.14, значение типа `double` может быть преобразовано в `int`. Следовательно, считать такой вызов ошибочным было бы слишком сурово. Вместо этого аргументы неявно преобразуются в `int` (отбрасыванием дробной части) и таким образом требования, налагаемые на типы параметров, выполняются. Поскольку при подобном преобразовании возможна потеря точности, хороший компилятор выдаст предупреждение. Вызов превращается в

```
gcd( 3, 6 );
```

что дает в результате `3`.

C++ является *строго типизированным* языком. Компилятор проверяет аргументы на соответствие типов в каждом вызове функции. Если тип фактического аргумента не соответствует типу формального параметра, то производится попытка неявного преобразования. Если же это оказывается невозможным или число аргументов неверно, компилятор выдает сообщение об ошибке. Именно поэтому функция должна быть объявлена до того, как программа впервые обратится к ней: без объявления компилятор не обладает информацией для проверки типов.

Пропуск аргумента при вызове или передача аргумента неуказанного типа часто служили источником ошибок в языке C. Теперь такие погрешности обнаруживаются на этапе компиляции.

Упражнение 7.1

```
(a) set( int *, int );
(b) void func();
(c) string error( int );
```

Какие из следующих прототипов функций содержат ошибки? Объясните.

```
(d) arr[10] sum( int *, int );
```

Упражнение 7.2

Напишите прототипы для следующих функций:

Функция с именем `compare`, имеющая два параметра типа ссылки на класс `matrix` и возвращающая значение типа `bool`.

Функция с именем `extract` без параметров, возвращающая контейнер `set` для хранения значений типа `int`. (Контейнерный тип `set` описывался в разделе 6.13.)

Упражнение 7.3

```
double calc( double );
int count( const string &, char );
void sum( vector<int> &, int );
```

Имеются объявления функций:

```
vector<int> vec( 10 );
```

Какие из следующих вызовов содержат ошибки и почему?

```
(a) calc( 23.4, 55.1 );
(b) count( "abcda", 'a' );
(c) sum( vec, 43.8 );

(d) calc( 66 );
```

7.3. Передача аргументов

Функции используют память из *стека программы*. Некоторая область стека отводится функции и остается связанной с ней до окончания ее работы, по завершении которой отведенная ей память освобождается и может быть занята другой функцией. Иногда эту часть стека называют *областью активации*.

Каждому параметру функции отводится место в данной области, причем его размер определяется типом параметра. При вызове функции память инициализируется значениями фактических аргументов.

Стандартным способом передачи аргументов является копирование их значений, т.е. *передача по значению*. При этом способе функция не получает доступа к реальным объектам, являющихся ее аргументами. Вместо этого она получает в стеке локальные копии этих объектов. Изменение значений копий никак не отражается на значениях самих объектов. Локальные копии теряются при выходе из функции.

Значения аргументов при передаче по значению не меняются. Следовательно, программист не должен заботиться о сохранении и восстановлении их значений при вызове функции. Без этого механизма любой вызов мог бы привести к нежелательному изменению аргументов, не объявленных константными явно. Передача по значению освобождает человека от лишних забот в наиболее типичной ситуации.

Однако такой способ передачи аргументов может не устраивать нас в следующих случаях:

- передача большого объекта типа класса. Временные и пространственные расходы на размещение и копирование такого объекта могут оказаться неприемлемыми для реальной программы;
- иногда значения аргументов должны быть модифицированы внутри функции. Например, `swap()` должна обменять значения своих аргументов, что невозможно

```
// swap() не меняет значений своих аргументов!
void swap( int vl, int v2 ) {
    int tmp = v2;
    v2 = vl;
    vl = tmp;
}
```

при передаче по значению:

`swap()` обменивает значения локальных копий своих аргументов. Те же переменные, что были использованы в качестве аргументов при вызове, остаются неизменными. Это можно проиллюстрировать, написав небольшую программу:

```
#include <iostream>
void swap( int, int );

int main() {
    int i = 10;
    int j = 20;

    cout << "Перед swap():\ti: "
         << i << "\tj: " << j << endl;

    swap( i, j );

    cout << "После swap():\ti: "
         << i << "\tj: " << j << endl;

    return 0;
}
```

Результат выполнения программы:

```
Перед swap():    i: 10    j: 20
После swap():    i: 10    j: 20
```

Достичь желаемого можно двумя способами. Первый – объявление параметров

```
// pswap() обменивает значения объектов,
// адресуемых указателями vl и v2
void pswap( int *vl, int *v2 ) {
    int tmp = *v2;
    *v2 = *vl;
    *vl = tmp;
}
```

указателями. Вот как будет выглядеть реализация `swap()` в этом случае:

Функция `main()` тоже нуждается в модификации. Вместо передачи самих объектов необходимо передавать их адреса:

```
pswap( &i, &j );
```

Теперь программа работает правильно:

```
Перед swap():    i: 10    j: 20
После swap():    i: 20    j: 10
```

Альтернативой может стать объявление параметров ссылками. В данном случае реализация `swap()` выглядит так:

```
// rswap() обменивает значения объектов,
// на которые ссылаются vl и v2
void rswap( int &vl, int &v2 ) {
    int tmp = v2;
    v2 = vl;
    vl = tmp;
}
```

Вызов этой функции из `main()` аналогичен вызову первоначальной функции `swap()`:

```
rswap( i, j );
```

Выполнив программу `main()`, мы снова получим верный результат.

7.3.1. Параметры-ссылки

Использование ссылок в качестве параметров модифицирует стандартный механизм передачи по значению. При такой передаче функция манипулирует локальными копиями аргументов. Используя параметры-ссылки, она получает l-значения своих аргументов и может изменять их.

В каких случаях применение параметров-ссылок оправданно? Во-первых, тогда, когда без использования ссылок пришлось бы менять типы параметров на указатели (см. приведенную выше функцию `swap()`). Во-вторых, при необходимости вернуть из функции несколько значений. В-третьих, для передачи большого объекта типа класса. Рассмотрим два последних случая подробнее.

Как пример функции, использующей параметр-ссылку для возврата дополнительного значения, возьмем `look_up()`, которая будет искать заданную величину в векторе целых чисел. В случае успеха `look_up()` вернет итератор, указывающий на найденный элемент, иначе – на элемент, расположенный за конечным. Если величина содержится в векторе несколько раз, итератор будет указывать на первое вхождение. Кроме того, дополнительный параметр-ссылка `occurs` возвращает количество найденных элементов.

```
#include <vector>

// параметр-ссылка 'occurs'
// содержит второе возвращаемое значение

vector<int>::const_iterator look_up(
    const vector<int> &vec,

    int value,     // искомое значение
    int &occurs )  // количество вхождений
{
    // res_iter инициализируется значением
    // следующего за конечным элемента
    vector<int>::const_iterator res_iter = vec.end();
    occurs = 0;

    for ( vector<int>::const_iterator iter = vec.begin();
             iter != vec.end();
             ++iter )
        if ( *iter == value )
        {
            if ( res_iter == vec.end() )
                res_iter = iter;
            ++occurs;
        }

    return res_iter;
}
```

Третий случай, когда использование параметра-ссылки может быть полезно, – это большой объект типа класса в качестве аргумента. При передаче по значению объект будет копироваться целиком при каждом вызове функции, что для больших объектов может привести к потере эффективности. Используя параметр-ссылку, функция получает доступ к той области памяти, где размещен сам объект, без создания дополнительной

```
class Huge { public: double stuff[1000]; };
extern int calc( const Huge & );

int main() {
    Huge table[ 1000 ];
    // ... инициализация table

    int sum = 0;
    for ( int ix=0; ix < 1000; ++ix )
        // calc() ссылается на элемент массива
        // типа Huge
        sum += calc( tab1e[ix] );
    // ...
}
```

копии. Например:

Может возникнуть желание использовать параметр-ссылку, чтобы избежать создания копии большого объекта, но в то же время не дать вызываемой функции возможности изменять значение аргумента. Если параметр-ссылка не должен модифицироваться внутри функции, то стоит объявить его как ссылку на константу. В такой ситуации

компилятор способен распознать и пресечь попытку непреднамеренного изменения значения аргумента.

В следующем примере нарушается константность параметра xx функции foo(). Поскольку параметр функции foo_bar() не является ссылкой на константу, то нет гарантии, что вызов foo_bar() не изменит значения аргумента. Компилятор

```
class X;
extern int foo_bar( X& );

int foo( const X& xx ) {
    // ошибка: константа передается
    // функции с параметром неконстантного типа
    return foo_bar( xx );
```

сигнализирует об ошибке:

```
}
```

Для того чтобы программа компилировалась, мы должны изменить тип параметра

```
extern int foo_bar( const X& );
```

foo_bar(). Подойдет любой из следующих двух вариантов:

```
extern int foo_bar( X ); // передача по значению
```

```
int foo( const X &xx ) {
    // ...
    X x2 = xx; // создать копию значения

    // foo_bar() может поменять x2,
    // xx останется нетронутым
    return foo_bar( x2 ); // правильно
```

Вместо этого можно передать копию xx, которую позволено менять:

```
}
```

Параметр-ссылка может именовать любой встроенный тип данных. В частности, разрешается объявить параметр как ссылку на указатель, если программист хочет изменить значение самого указателя, а не объекта, который он адресует. Вот пример

```
void ptrswap( int *&vl, int *&v2 ) {
    int *trnp = v2;
    v2 = vl;
    vl = tmp;
```

функции, обменивающей друг с другом значения двух указателей:

```
}
```

Объявление

```
int *&v1;
```

должно читаться справа налево: v1 является ссылкой на указатель на объект типа int. Модифицируем функцию main(), которая вызывала rswap(), для проверки работы

```
#include <iostream>
void ptrswap( int *&vl, int *&v2 );

int main() {
    int i = 10;
    int j = 20;

    int *pi = &i;
    int *pj = &j;

    cout << "Перед ptrswap():\tpi: "
         << *pi << "\tpj: " << *pj << endl;

    ptrswap( pi, pj );
    cout << "После ptrswap():\tpi: "
         << *pi << "\tpj: " << pj << endl;

    return 0;
```

ptrswap():

```
}
```

Вот результат работы программы:

```
Перед ptrswap():    pi: 10    pj: 20
После ptrswap():    pi: 20    pj: 10
```

7.3.2. Параметры-ссылки и параметры-указатели

Когда же лучше использовать параметры-ссылки, а когда – параметры-указатели? В конце концов, и те и другие позволяют функции модифицировать объекты, эффективно передавать в функцию большие объекты типа класса. Что выбрать: объявить параметр ссылкой или указателем?

Как было сказано в разделе 3.6, ссылка может быть один раз инициализирована значением объекта, и впоследствии изменить ее нельзя. Указатель же в течение своей жизни способен адресовать разные объекты или не адресовать вообще.

Поскольку указатель может содержать, а может и не содержать адрес какого-либо

```
class X;
void manip( X *px )
{
    // проверим на 0 перед использованием
    if ( px != 0 )
        // обратимся к объекту по адресу...
```

объекта, перед его использованием функция должна проверить, не равен ли он нулю:

```
}
```

Параметр-ссылка не нуждается в этой проверке, так как всегда существует именуемый ею

```
class Type { };
void operate( const Type& p1, const Type& p2 );

int main() {
    Type obj1;
    // присвоим objl некоторое значение

    // ошибка: ссылка не может быть равной 0
    Type obj2 = operate( objl, 0 );
}
```

объект. Например:

Если параметр должен ссылаться на разные объекты во время выполнения функции или принимать нулевое значение (ни на что не ссылаться), нам следует использовать указатель.

Одна из важнейших сфер применения параметров-ссылок – эффективная реализация перегруженных операций. При этом использование операций остается простым и интуитивно понятным. (Подробнее данный вопрос рассматривается в главе 15.) Разберем маленький пример. Представим себе класс `Matrix` (матрица). Хорошо бы реализовать

```
Matrix a, b, c;
```

операции сложения и присваивания “привычным” способом:

```
c = a + b;
```

Эти операции реализуются с помощью перегруженных операторов – функций с немного необычным именем. Для оператора сложения такая функция будет называться

```
Matrix       // тип возврата -  Matrix
operator+(   // имя перегруженного оператора
Matrix m1,   // тип левого операнда
Matrix m2    // тип правого операнда
)
{
    Matrix result;
    // необходимые действия
    return result;
}
```

`operator+`. Посмотрим, как ее определить:

При такой реализации сложение двух объектов типа `Matrix` выглядит вполне привычно:

```
a + b;
```

но, к сожалению, оказывается совершенно неэффективным. Заметим, что параметры у нас передаются по значению. Содержимое двух матриц будет копироваться в область активации функции `operator+()`, а поскольку объекты типа `Matrix` весьма велики, затраты времени и памяти на создание копий могут быть совершенно неприемлемыми.

Представим себе, что мы решили использовать указатели в качестве параметров, чтобы

```
// реализация с параметрами-указателями
operator+( Matrix *ml, Matrix *m2 )
{
    Matrix result;
    // необходимые действия
    return result;
}
```

избежать этих затрат. Вот модифицированный код `operator+()`:

Да, мы добились эффективной реализации, но зато теперь применение нашей операции вряд ли можно назвать интуитивно понятным. В качестве значений параметров-указателей требуется передавать адреса складываемых объектов. Поэтому для сложения двух матриц пришлось бы написать:

```
&a + &b; // допустимо, хотя и плохо
```

Хотя такая форма не может не вызвать критику, но все-таки два объекта сложить еще

```
// а вот это не работает
// &a + &b возвращает объект типа Matrix
```

удается. А вот три уже крайне затруднительно:

```
&a + &b + &c;
```

```
// правильно: работает, однако ...
```

Для того чтобы сложить три объекта, при подобной реализации нужно написать так:

```
&( &a + &b ) + &c;
```

Трудно ожидать, что кто-нибудь согласится писать такие выражения. К счастью, параметры-ссылки дают именно то решение, которое требуется. Если параметр объявлен как ссылка, функция получает его l-значение, а не копию. Лишнее копирование исключается. И тип фактического аргумента может быть `Matrix` – это упрощает операцию сложения, как и для встроенных типов. Вот схема перегруженного оператора

```
// реализация с параметрами-ссылками
operator+( const Matrix &m1, const Matrix &m2 )
{
    Matrix result;
    // необходимые действия
    return result;
}
```

сложения для класса `Matrix`:

При такой реализации сложение трех объектов `Matrix` выглядит вполне привычно:

```
a + b + c;
```

Ссылки были введены в C++ именно для того, чтобы удовлетворить двум требованиям: эффективная реализация и интуитивно понятное применение.

7.3.3. Параметры-массивы

Массив в C++ никогда не передается по значению, а только как указатель на его первый, точнее нулевой, элемент. Например, объявление

```
void putValues( int[ 10 ] );
```

рассматривается компилятором так, как будто оно имеет вид

```
void putValues( int* );
```

Размер массива неважен при объявлении параметра. Все три приведенные записи

```
// три эквивалентных объявления putValues()
void putValues( int* );
void putValues( int[] );
```

эквивалентны:

```
void putValues( int[ 10 ] );
```

Передача массивов как указателей имеет следующие особенности:

- изменение значения аргумента внутри функции затрагивает сам переданный объект, а не его локальную копию. Если такое поведение нежелательно, программист должен позаботиться о сохранении исходного значения. Можно также при объявлении функции указать, что она не должна изменять значение параметра, объявив этот параметр константой:

```
void putValues( const int[ 10 ] );
```

- размер массива не является частью типа параметра. Поэтому функция не знает реального размера передаваемого массива. Компилятор тоже не может это

```
void putValues( int[ 10 ] ); // рассматривается как int*
int main() {
    int i, j [ 2 ];
    putValues( &i ); // правильно: &i is int*;
                     // однако при выполнении возможна ошибка
    putValues( j ); // правильно: j - адрес 0-го элемента - int*;
```

проверить. Рассмотрим пример:

```
                     // однако при выполнении возможна ошибка
```

При проверке типов параметров компилятор способен распознать, что в обоих случаях тип аргумента `int*` соответствует объявлению функции. Однако контроль за тем, не является ли аргумент массивом, не производится.

По принятому соглашению C-строка является массивом символов, последний элемент которого равен нулю. Во всех остальных случаях при передаче массива в качестве параметра необходимо указывать его размер. Это относится и к массивам символов, внутри которых встречается 0. Обычно для такого указания используют дополнительный

```
void putValues( int[], int size );
int main() {
    int i, j[ 2 ];
    putValues( &i, 1 );
    putValues( j, 2 );
    return 0;
}
```

параметр функции. Например:

putValues() печатает элементы массива в следующем формате:

```
( 10 )< 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 >
```

где 10 – это размер массива. Вот как выглядит реализация putValues(), в которой

```
#include <iostream>

const lineLength =12; // количество элементов в строке
void putValues( int *ia, int sz )
{
    cout << "( " << sz << " )< ";
    for (int i=0;i<sz; ++i )
    {
        if ( i % lineLength == 0 && i )
            cout << "\n\t"; // строка заполнена

        cout << ia[ i ];

        // разделитель, печатаемый после каждого элемента,
        // кроме последнего
        if ( i % lineLength != lineLength-1 &&
                i != sz-1 )
            cout << ", ";
    }
    cout << " >\n";
}
```

используется дополнительный параметр:

Другой способ сообщить функции размер массива-параметра – объявить параметр как ссылку. В этом случае размер становится частью типа, и компилятор может проверить аргумент в полной мере.

```
// параметр - ссылка на массив из 10 целых
void putValues( int (&arr)[10] );
int main() {
    int i, j [ 2 ];
    putValues(i); // ошибка:
                  // аргумент не является массивом из 10 целых
    putValues(j); // ошибка:
                  // аргумент не является массивом из 10 целых
    return 0;
}
```

Поскольку размер массива теперь является частью типа параметра, новая версия putValues() способна работать только с массивами из 10 элементов. Конечно, это

```
#include <iostream>

void putValues( int (&ia)[10] )
{
    cout << "( 10 )< ";
    for ( int 1 =0; i < 10; ++i ) { cout << ia[ i ];

    // разделитель, печатаемый после каждого элемента,
    // кроме последнего
    if ( i != 9 )
        cout << ", ";
    }
    cout << " >\n";
}
```

ограничивает ее область применения, зато реализация значительно проще:

Еще один способ получить размер переданного массива в функции – использовать абстрактный контейнерный тип. (Такие типы были представлены в главе 6. В следующем подразделе мы поговорим об этом подробнее.)

Хотя две предыдущих реализации putValues() правильны, они обладают серьезными недостатками. Так, первый вариант работает только с массивами типа int. Для типа double* нужно писать другую функцию, для long* – еще одну и т.д. Второй вариант производит операции только над массивом из 10 элементов типа int. Для обработки массивов разного размера нужны дополнительные функции. Лучшим решением было бы использовать шаблон – функцию, или, скорее, обобщенную реализацию кода целого семейства функций, которые отличаются только типами обрабатываемых данных. Вот как можно сделать из первого варианта putValues() шаблон, способный работать с

```
template <class Type>
void putValues( Type *ia, int sz )
{
    // так же, как и раньше
}
```

массивами разных типов и размеров:

Параметры шаблона заключаются в угловые скобки. Ключевое слово class означает, что идентификатор Type служит именем параметра, при конкретизации шаблона функции

putValues() он заменяется на реальный тип – int, double, string и т.д. (В главе 10 мы продолжим разговор о шаблонах функций.)

Параметр может быть многомерным массивом. Для такого параметра должны быть заданы правые границы всех измерений, кроме первого. Например:

```
putValues( int matrix[][10], int rowSize );
```

Здесь matrix объявляется как двумерный массив, который содержит десять столбцов и неизвестное число строк. Эквивалентным объявлением для matrix будет:

```
int (*matrix)[10]
```

Многомерный массив передается как указатель на его нулевой элемент. В нашем случае тип matrix – указатель на массив из десяти элементов типа int. Как и для одномерного массива, граница первого измерения не учитывается при проверке типов. Если параметры являются многомерными массивами, то контролируются все измерения, кроме первого.

Заметим, что скобки вокруг *matrix необходимы из-за более высокого приоритета операции взятия индекса. Инструкция

```
int *matrix[10];
```

объявляет matrix как массив из десяти указателей на int.

7.3.4. Абстрактные контейнерные типы в качестве параметров

Абстрактные контейнерные типы, представленные в главе 6, также используются для объявления параметров функции. Например, можно определить putValues() как имеющую параметр типа vector<int> вместо встроенного типа массива.

Контейнерный тип является классом и обеспечивает значительно большую функциональность, чем встроенные массивы. Так, vector<int> “знает” собственный размер. В предыдущем подразделе мы видели, что размер параметра-массива неизвестен функции и для его передачи приходится задавать дополнительный параметр. Использование vector<int> позволяет обойти это ограничение. Например, можно изменить определение нашей putValues() на такое:

```
#include <iostream>
#include <vector>

const lineLength =12; // количество элементов в строке
void putValues( vector<int> vec )
{
    cout << "( " << vec.size() << " )< ";
    for ( int i = 0; i < vec.size(); ++1 ) {
        if ( i % lineLength == 0 && i )
            cout << "\n\t"; // строка заполнена

    cout << vec[ i ];

    // разделитель, печатаемый после каждого элемента,
    // кроме последнего
    if ( 1 % lineLength != lineLength-1 &&
                i != vec.size()-1 )
            cout << ", ";
    }
    cout << " >\n";

}
```

```
void putValues( vector<int> );
int main() {
    int i, j[ 2 ];
    // присвоить i и j некоторые значения
    vector<int> vec1(1); // создадим вектор из 1 элемента
    vecl[0] = i;
    putValues( vecl );

    vector<int> vec2;    // создадим пустой вектор
    // добавим элементы к vec2
    for ( int ix = 0;
           ix < sizeof( j ) / sizeof( j[0] );
           ++ix )
      // vec2[ix] == j [ix]
      vec2.push_back( j[ix] );
    putValues( vec2 );

    return 0;
```

Функция `main()`, вызывающая нашу новую функцию `putValues()`, выглядит так:

```
}
```

Заметим, что параметр `putValues()`передается по значению. В подобных случаях контейнер со всеми своими элементами всегда копируется в стек вызванной функции. Поскольку операция копирования весьма неэффективна, такие параметры лучше объявлять как ссылки.

Как бы вы изменили объявление `putValues()`?

Вспомним, что если функция не модифицирует значение своего параметра, то предпочтительнее, чтобы он был ссылкой на константный тип:

```
void putValues( const vector<int> & ) { ...
```

7.3.5. Значения параметров по умолчанию

Значение параметра по умолчанию – это значение, которое разработчик считает подходящим в большинстве случаев употребления функции, хотя и не во всех. Оно освобождает программиста от необходимости уделять внимание каждой детали интерфейса функции.

Значения по умолчанию для одного или нескольких параметров функции задаются с помощью того же синтаксиса, который употребляется при инициализации переменных. Например, функция для создания и инициализации двумерного массива, моделирующего экран терминала, может использовать такие значения для высоты, ширины и символа

```
char *screenInit( int height = 24, int width = 80,
```

фона экрана:

```
                  char background = ' ' );
```

Функция, для которой задано значение параметра по умолчанию, может вызываться по-разному. Если аргумент опущен, используется значение по умолчанию, в противном случае – значение переданного аргумента. Все следующие вызовы `screenInit()`

```
char *cursor;

// эквивалентно screenInit(24,80,' ')
cursor = screenInit();

// эквивалентно screenInit(66,80,' ')
cursor = screenlnit(66);

// эквивалентно screenInit(66,256,' ')
cursor = screenlnit(66, 256);
```

корректны:

```
cursor = screenlnit(66, 256, '#');
```

Фактические аргументы сопоставляются с формальными параметрами позиционно (в порядке следования), и значения по умолчанию могут использоваться только для подстановки вместо отсутствующих последних аргументов. В нашем примере

```
// эквивалентно screenInit('?',80,' ')
cursor = screenInit('?');

// ошибка, неэквивалентно screenInit(24,80,'?')
```

невозможно задать значение для `background`, не задавая его для `height` и `width`.

```
cursor = screenInit( , ,'?');
```

При разработке функции с параметрами по умолчанию придется позаботиться об их расположении. Те, для которых значения по умолчанию вряд ли будут употребляться, необходимо поместить в начало списка. Функция `screenInit()` предполагает (возможно, основываясь на опыте применения), что параметр `height` будет востребован пользователем наиболее часто.

Значения по умолчанию могут задаваться для всех параметров или только для некоторых. При этом параметры без таких значений должны идти раньше тех, для которых они

```
// ошибка: width должна иметь значение по умолчанию,
// если такое значение имеет height
char *screenlnit( int height = 24, int width,
```

указаны.

```
                  char background = ' ' );
```

Значение по умолчанию может указываться только один раз в файле. Следующая запись

```
// tf.h
int ff( int = 0 );

// ft.C
#include "ff.h"
```

ошибочна:

```
int ff( int i = 0) { ... } // ошибка
```

По соглашению значение задается в объявлении функции, которое размещается в общедоступном заголовочном файле (описывающем интерфейс), а не в ее определении. Если же указать его в определении, это значение будет доступно только для вызовов функции внутри исходного файла, содержащего это определение.

Можно объявить функцию повторно и таким образом задать дополнительные параметры по умолчанию. Это удобно при настройке универсальной функции для конкретного приложения. Скажем, в системной библиотеке UNIX есть функция `chmod()`, изменяющая режим доступа к файлу. Ее объявление содержится в системном заголовочном файле `<cstdlib>`:

```
int chmod( char *filePath, int protMode );
```

`protMode` представляет собой режим доступа, а `filePath` – имя и каталог файла. Если в некотором приложении файл только читается, можно переобъявить функцию `chmod()`, задав для соответствующего параметра значение по умолчанию, чтобы не указывать его

```
#include <cstdlib>
```

при каждом вызове:

```
int chmod( char *filePath, int protMode=0444 );
```

Если функция объявлена в заголовочном файле так:

```
file int ff( int a, int b, int c = 0 ); // ff.h
```

то как переобъявить ее, чтобы присвоить значение по умолчанию для параметра `b`? Следующая строка ошибочна, поскольку она повторно задает значение для `c`:

```
#include "ff.h"
int ff( int a, int b = 0, int c = 0 ); // ошибка
```

```
#include "ff.h"
```

Так выглядит правильное объявление:

```
int ff( int a, int b = 0, int c ); // правильно
```

В том месте, где мы переобъявляем функцию `ff()`, параметр `b` расположен правее других, не имеющих значения по умолчанию. Поэтому требование присваивать такие

```
#include "ff.h"
int ff( int a, int b = 0, int c ); // правильно
```

значения справа налево не нарушается. Теперь мы можем переобъявить `ff()` еще раз:

```
int ff( int a = 0, int b, int c ); // правильно
```

Значение по умолчанию не обязано быть константным выражением, можно использовать

```
int aDefault();
int bDefault( int );
int cDefault( double = 7.8 );

int glob;

int ff( int a = aDefault() ,
        int b = bDefau1t( glob ) ,
```

любое:

```
        int c = cDefault() );
```

Если такое значение является выражением, то оно вычисляется во время вызова функции. В примере выше `cDefault()` работает каждый раз, когда происходит вызов функции `ff()` без указания третьего аргумента.

7.3.6. Многоточие

Иногда нельзя перечислить типы и количество всех возможных аргументов функции. В этих случаях список параметров представляется многоточием (`...`), которое отключает механизм проверки типов. Наличие многоточия говорит компилятору, что у функции может быть произвольное количество аргументов неизвестных заранее типов.

```
void foo( parm_list, ... );
```

Многоточие употребляется в двух форматах:

```
void foo( ... );
```

Первый формат предоставляет объявления для части параметров. В этом случае проверка типов для объявленных параметров производится, а для оставшихся фактических аргументов – нет. Запятая после объявления известных параметров необязательна.

Примером вынужденного использования многоточия служит функция `printf()` стандартной библиотеки C. Ее первый параметр является C-строкой:

```
int printf( const char* ... );
```

Это гарантирует, что при любом вызове `printf()` ей будет передан первый аргумент типа `const char*`. Содержание такой строки, называемой *форматной*, определяет, необходимы ли дополнительные аргументы при вызове. При наличии в строке формата метасимволов, начинающихся с символа %, функция ждет присутствия этих аргументов. Например, вызов

```
printf( "hello, world\n" );
```

имеет один строковый аргумент. Но

```
printf( "hello, %s\n", userName );
```

имеет два аргумента. Символ % говорит о наличии второго аргумента, а буква `s`, следующая за ним, определяет его тип – в данном случае символьную строку.

Большинство функций с многоточием в объявлении получают информацию о типах и количестве фактических параметров по значению явно объявленного параметра. Следовательно, первый формат многоточия употребляется чаще.

```
void f();
```

Отметим, что следующие объявления неэквивалентны:

```
void f( ... );
```

В первом случае `f()` объявлена как функция без параметров, во втором – как имеющая

```
f( someValue );
```

ноль или более параметров. Вызовы

```
f( cnt, a, b, c );
```

корректны только для второго объявления. Вызов

```
f();
```

применим к любой из двух функций.

Упражнение 7.4

Какие из следующих объявлений содержат ошибки? Объясните.

```
(a) void print( int arr[][], int size );
(b) int ff( int a, int b = 0, int c = 0 );

(d)       char   *screenInit( int height = 24, int width,
                              char  background );
(c) void operate( int *matrix[] );
(e)       void   putValues( int (&ia)[] );
```

Упражнение 7.5

```
(a) char *screenInit( int height, int width,
                      char background = ' ' );
    char *screenInit( int height = 24, int width,
                      char background );

(b) void print( int (*arr)[6], int size );
    void print( int (*arr)[5], int size );

(c) void manip( int *pi, int first, int end = 0 );
```

Повторные объявления всех приведенных ниже функций содержат ошибки. Найдите их.

```
    void manip( int *pi, int first = 0, int end = 0 );
```

Упражнение 7.6

```
void print( int arr[][5], int size );
void operate(int *matrix[7]);
char *screenInit( int height = 24, int width = 80,
                  char background = ' ' );
```

Даны объявления функций.

```
(a) screenInit();

(b) int *matrix[5];
    operate( matrix );

(c) int arr[5][5];
```

Вызовы этих функций содержат ошибки. Найдите их и объясните.

```
    print( arr, 5 );
```

Упражнение 7.7

Перепишите функцию `putValues( vector<int> )`, приведенную в подразделе 7.3.4, так, чтобы она работала с контейнером `list<string>`. Печатайте по одному значению на строке. Вот пример вывода для списка из двух строк:

```
( 2 )
```

```
<
"first string"
"second string"
>
```

Напишите функцию `main()`, вызывающую новый вариант `putValues()` со следующим

```
"put function declarations in header files"
"use abstract container types instead of built-in arrays"
"declare class parameters as references"
"use reference to const types for invariant parameters"
```

списком строк:

```
"use less than eight parameters"
```

Упражнение 7.8

В каком случае вы применили бы параметр-указатель? А в каком – параметр-ссылку? Опишите достоинства и недостатки каждого способа.

7.4. Возврат значения

В теле функции может встретиться инструкция `return`. Она завершает выполнение функции. После этого управление возвращается той функции, из которой была вызвана

```
return;
```

данная. Инструкция `return` может употребляться в двух формах:

```
return expression;
```

Первая форма используется в функциях, для которых типом возвращаемого значения является `void`. Использовать `return` в таких случаях обязательно, если нужно принудительно завершить работу. (Такое применение `return` напоминает инструкцию `break`, представленную в разделе 5.8.) После конечной инструкции функции подразумевается наличие `return`. Например:

```
void d_copy( double "src, double *dst, int sz )
{
    /* копируем массив "src" в "dst"
    * для простоты предполагаем, что они одного размера
    */

    // завершение, если хотя бы один из указателей равен 0
    if ( !src || !dst )
        return;

    // завершение,
    // если указатели адресуют один и тот же массив
    if ( src == dst )
        return;

    // копировать нечего
    if ( sz == 0 )
        return;

    // все еще не закончили?
    // тогда самое время что-то сделать
    for ( int ix = 0; ix < sz; ++ix )
        dst[ix] = src[ix];

    // явного завершения не требуется
}
```

Во второй форме инструкции return указывается то значение, которое функция должна вернуть. Это значение может быть сколь угодно сложным выражением, даже содержать вызов функции. В реализации функции factorial(), которую мы рассмотрим в следующем разделе, используется return следующего вида:

```
return val * factorial(val-1);
```

В функции, не объявленная с void в качестве типа возвращаемого значения, обязательно использовать вторую форму return, иначе произойдет ошибка компиляции. Хотя компилятор не отвечает за правильность результата, он сможет гарантировать его наличие. Следующая программа не компилируется из-за двух мест, где программа завершается без возврата значения:

```
// определение интерфейса класса Matrix
#include "Matrix.h"

bool is_equa1( const Matrix &ml, const Matrix &m2 )
{
    /* Если содержимое двух объектов Matrix одинаково,
    *  возвращаем true;
    *  в противном случае - false
    */

    // сравним количество столбцов
    if ( ml.colSize() != m2.co1Size() )
        // ошибка: нет возвращаемого значения
        return;

    // сравним количество строк
    if ( ml.rowSize() != m2.rowSize() )
        // ошибка: нет возвращаемого значения
        return;

    // пробежимся по обеим матрицам, пока
    // не найдем неравные элементы
    for ( int row = 0; row < ml.rowSize(); ++row )
        for ( int col = 0; co1 < ml.colSize(); ++co1 )
           if ( ml[row][col] != m2[row][col] )
               return false;
    // ошибка: нет возвращаемого значения
    // для случая равенства
}
```

Если тип возвращаемого значения не точно соответствует указанному в объявлении функции, то применяется неявное преобразование типов. Если же стандартное приведение невозможно, происходит ошибка компиляции. (Преобразования типов рассматривались в разделе 4.1.4.)

По умолчанию возвращаемое значение *передается по значению*, т.е. вызывающая функция получает копию результата вычисления выражения, указанного в инструкции

```
Matrix grow( Matrix* p ) {
    Matrix val;
    // ...
    return val;
}
```

`return`. Например:

`grow()` возвращает вызывающей функции копию значения, хранящегося в переменной `val`.

Такое поведение можно изменить, если объявить, что возвращается указатель или ссылка. При возврате ссылки вызывающая функция получает l-значение для `val` и потому может модифицировать `val` или взять ее адрес. Вот как можно объявить, что `grow()` возвращает ссылку:

```
Matrix& grow( Matrix* p ) {
    Matrix *res;
    // выделим память для объекта Matrix
    // большого размера
    // res адресует этот новый объект
    // скопируем содержимое *p в *res
    return *res;
}
```

Если возвращается большой объект, то гораздо эффективнее перейти от возврата по значению к использованию ссылки или указателя. В некоторых случаях компилятор может сделать это автоматически. Такая оптимизация получила название *именованное возвращаемое значение.* (Она описывается в разделе 14.8.)

Объявляя функцию как возвращающую ссылку, программист должен помнить о двух возможных ошибках:

- возврат ссылки на локальный объект, время жизни которого ограничено временем выполнения функции. (О времени жизни локальных объектов речь пойдет в разделе 8.3.) По завершении функции такой ссылке соответствует область

```
// ошибка: возврат ссылки на локальный объект
Matrix& add( Matrix &m1, Matrix &m2 )
{
    Matrix result:
    if ( m1.isZero()   )
        return m2;
    if ( m2.isZero()   )
        return m1;

    // сложим содержимое двух матриц
    // ошибка: ссылка на сомнительную область памяти
    // после возврата
    return result;
}
```

  памяти, содержащая неопределенное значение. Например:

  В таком случае тип возврата не должен быть ссылкой. Тогда локальная переменная может быть скопирована до окончания времени своей жизни:

```
Matrix add( ... )
```

- функция возвращает l-значение. Любая его модификация затрагивает сам объект. Например:

```
#include <vector>

int &get_val( vector<int> &vi, int ix ) {
    return vi [ix];
}

int ai[4] = { 0, 1, 2, 3 };
vector<int> vec( ai, ai+4 ); // копируем 4 элемента ai в vec

int main() {
    // увеличивает vec[0] на 1
    get_val( vec.0 )++;
    // ...
}
```

Для предотвращения нечаянной модификации возвращенного объекта нужно объявить тип возврата как `const`:

```
const int &get_val( ... )
```

Примером ситуации, когда l-значение возвращается намеренно, чтобы позволить модифицировать реальный объект, может служить перегруженный оператор взятия индекса для класса `IntArray` из раздела 2.3.

7.4.1. Передача данных через параметры и через глобальные объекты

Различные функции программы могут общаться между собой с помощью двух механизмов. (Под словом “общаться” мы подразумеваем обмен данными.) В одном случае используются глобальные объекты, в другом – передача параметров и возврат значений.

```
int glob;
int main() {
   // что угодно
}
```

Глобальный объект определен вне функции. Например:

Объект `glob` является глобальным. (В главе 8 рассмотрение глобальных объектов и глобальной области видимости будет продолжено.) Главное достоинство и одновременно один из наиболее заметных недостатков такого объекта – доступность из любого места программы, поэтому его обычно используют для общения между разными модулями. Обратная сторона медали такова:

- функции, использующие глобальные объекты, зависят от этих объектов и их типов. Использовать такую функцию в другом контексте затруднительно;
- при модификации такой программы повышается вероятность ошибок. Даже для внесения локальных изменений необходимо понимание всей программы в целом;

- если глобальный объект получает неверное значение, ошибку нужно искать по всей программе. Отсутствует локализация;
- используя глобальные объекты, труднее писать рекурсивные функции (Рекурсия возникает тогда, когда функция вызывает сама себя. Мы рассмотрим это в разделе 7.5.);
- если используются *потоки* (*threads*), то для синхронизации доступа к глобальным объектам требуется писать дополнительный код. Отсутствие синхронизации – одна из распространенных ошибок при использовании потоков. (Пример использования потоков при программировании на C++ см. в статье “Distributing Object Computing in C++” (Steve Vinoski and Doug Schmidt) в [LIPPMAN96b].)

Можно сделать вывод, что для передачи информации между функциями предпочтительнее пользоваться параметрами и возвращаемыми значениями.

Вероятность ошибок при таком подходе возрастает с увеличением списка. Считается, что восемь параметров – это приемлемый максимум. В качестве альтернативы длинному списку можно использовать в качестве параметра класс, массив или контейнер. Он способен содержать группу значений.

Аналогично программа может возвращать только одно значение. Если же логика требует нескольких, некоторые параметры объявляются ссылками, чтобы функция могла непосредственно модифицировать значения соответствующих фактических аргументов и использовать эти параметры для возврата дополнительных значений, либо некоторый класс или контейнер, содержащий группу значений, объявляется типом, возвращаемым функцией.

Упражнение 7.9

Каковы две формы инструкции `return`? Объясните, в каких случаях следует использовать первую, а в каких вторую форму.

Упражнение 7.10

```
vector<string> &readText( ) {
    vector<string> text;

    string word;
    while ( cin >> word ) {
        text.push_back( word );
        // ...
    }
    // ....
    return text;
```

Найдите в данной функции потенциальную ошибку времени выполнения:

```
}
```

Упражнение 7.11

Каким способом вы вернули бы из функции несколько значений? Опишите достоинства и недостатки вашего подхода.

7.5. Рекурсия

Функция, которая прямо или косвенно вызывает сама себя, называется *рекурсивной*.

```
int rgcd( int vl, int v2 )
{
    if ( v2 != 0 )
        return rgcd( v2, vl%v2 );
    return vl;
}
```

Например:

Такая функция обязательно должна определять условие окончания, в противном случае рекурсия будет продолжаться бесконечно. Подобную ошибку так иногда и называют – *бесконечная рекурсия*. Для `rgcd()` условием окончания является равенство нулю остатка.

Вызов

```
rgcd( 15, 123 );
```

возвращает 3 (см. табл. 7.1).

Таблица 7.1. Трассировка вызова rgcd (15,123)

vl	v2	return
15	123	rgcd(123,15)
123	15	rgcd(15,3)
15	3	rgcd(3,0)
3	0	3

Последний вызов,

```
rgcd(3,0);
```

удовлетворяет условию окончания. Функция возвращает наибольший общий делитель, он же возвращается и каждым предшествующим вызовом. Говорят, что значение *всплывает* (percolates) вверх, пока управление не вернется в функцию, вызвавшую `rgcd()` в первый раз.

Рекурсивные функции обычно выполняются медленнее, чем их нерекурсивные (итеративные) аналоги. Это связано с затратами времени на вызов функции. Однако, как правило, они компактнее и понятнее.

Приведем пример. Факториалом числа `n` является произведение натуральных чисел от `1` до `n`. Так, факториал `5` равен `120`: $1 \times 2 \times 3 \times 4 \times 5 = 120$.

Вычислять факториал удобно с помощью рекурсивной функции:

```
unsigned long
factorial( int val ) {
    if ( val > 1 )
        return val * factorial( val-1 );
    return 1;
}
```

Рекурсия обрывается по достижении `val` значения 1.

Упражнение 7.12

Перепишите `factorial()` как итеративную функцию.

Упражнение 7.13

Что произойдет, если условием окончания `factorial()` будет следующее:

```
if ( val != 0 )
```

7.6. Встроенные функции

```
int min( int vl, int v2 )
{
    return( vl < v2 ? vl : v2 );
}
```

Рассмотрим следующую функцию `min()`:

Преимущества определения функции для такой небольшой операции таковы:

- как правило, проще прочесть и интерпретировать вызов `min()`, чем читать условный оператор и вникать в смысл его действий, особенно если `v1` и `v2` являются сложными выражениями;
- модифицировать одну локализованную реализацию в приложении легче, чем 300. Например, если будет решено изменить проверку на:

```
( vl == v2 || vl < v2 )
```

  поиск каждого ее вхождения будет утомительным и с большой долей вероятности приведет к ошибкам;

- семантика единообразна. Все проверки выполняются одинаково;
- функция может быть повторно использована в другом приложении.

Однако этот подход имеет один недостаток: вызов функции происходит медленнее, чем непосредственное вычисление условного оператора. Необходимо скопировать два аргумента, запомнить содержимое машинных регистров и передать управление в другое место программы. Решение дают встроенные функции. Встроенная функция “подставляется по месту” в каждой точке своего вызова. Например:

```
int minVa12 = min( i, j );
```

заменяется при компиляции на

```
int minVal2 = i < j ? i : j;
```

Таким образом, не требуется тратить время на реализацию `min()` в виде функции.

Функция `min()` объявляется как встроенная с помощью ключевого слова `inline` перед типом возвращаемого значения в объявлении или определении:

```
inline int min( int vl, int v2 ) { /* ... */ }
```

Заметим, однако, что спецификация `inline` – это только подсказка компилятору. Компилятор может проигнорировать ее, если функция плохо подходит для встраивания по месту. Например, рекурсивная функция (такая, как `rgcd()`) не может быть полностью встроена в месте вызова (хотя для самого первого вызова это возможно). Функция из 1200 строк также скорее всего не подойдет. В общем случае такой механизм предназначен для оптимизации небольших, простых, часто используемых функций. Он крайне важен для поддержки концепции сокрытия информации при разработке абстрактных типов данных. Например, встроенной объявлена функция-член `size()` в классе `IntArray` из раздела 2.3.

Встроенная функция должна быть видна компилятору в месте вызова. В отличие от обычной, такая функция определяется в каждом исходном файле, где есть обращения к ней. Конечно же, определения одной и той же встроенной функции в разных файлах должны совпадать. Если программа содержит два исходных файла `compute.C` и `draw.C`, не нужно писать для них разные реализации функции `min()`. Если определения функции различаются, программа становится нестабильной: неизвестно, какое из них будет выбрано для каждого вызова, если компилятор не стал встраивать эту функцию.

Рекомендуется помещать определение встроенной функции в заголовочный файл и включать его во все файлы, где есть обращения к ней. Такой подход гарантирует, что для встроенной функции существует только одно определение и код не дублируется; дублирование может привести к непреднамеренному расхождению текстов в течение жизненного цикла программы.

Поскольку `min()` является общеупотребительной операцией, реализация ее входит в стандартную библиотеку C++; это один из обобщенных алгоритмов, описанных в главе 12 и в Приложении. Функция `min()` реализована как шаблон, что позволяет ей работать с операндами арифметического типа, отличного от `int`. (Шаблоны функций рассматриваются в главе 10.)

7.7. Директива связывания extern "C" A

Если программист хочет использовать функцию, написанную на другом языке, в частности на C, то компилятору нужно указать, что при вызове требуются несколько иные условия. Скажем, имя функции или порядок передачи аргументов различаются в зависимости от языка программирования.

Показать, что функция написана на другом языке, можно с помощью *директивы связывания* в форме *простой* либо *составной инструкции*:

```
// директива связывания в форме простой инструкции
extern "C" void exit(int);

// директива связывания в форме составной инструкции
extern "C" {
    int printf( const char* ... );
    int scanf( const char* ... );
}
// директива связывания в форме составной инструкции
extern "C" {
#include <cmath>
}
```

Первая форма такой директивы состоит из ключевого слова `extern`, за которым следует строковый литерал, а за ним – "обычное" объявление функции. Хотя функция написана на другом языке, проверка типов вызова выполняется полностью. Несколько объявлений функций могут быть помещены в фигурные скобки составной инструкции директивы связывания – второй формы этой директивы. Скобки отмечают те объявления, к которым она относится, не ограничивая их видимости, как в случае обычной составной инструкции. Составная инструкция `extern "C"` в предыдущем примере говорит только о том, что функции `printf()` и `scanf()` написаны на языке C. Во всех остальных отношениях эти объявления работают точно так же, как если бы они были расположены вне инструкции.

Если в фигурные скобки составной директивы связывания помещается директива препроцессора `#include`, все объявленные во включаемом заголовочном файле функции рассматриваются как написанные на языке, указанном в этой директиве. В предыдущем примере все функции из заголовочного файла `cmath` написаны на языке C.

Директива связывания не может появиться внутри тела функции. Следующий фрагмент

```
int main() {
    // ошибка: директива связывания не может появиться
    // внутри тела функции
    extern "C" double sqrt( double );
    double getValue(); //правильно

    double result = sqrt ( getValue() );
    //...
    return 0;
}
```

кода вызывает ошибку компиляции:

Если мы переместим директиву так, чтобы она оказалась вне тела `main()`, программа

```
extern "C" double sqrt( double );
int main() {
    double getValue(); //правильно

    double result = sqrt ( getValue() );
    //...
    return 0;
}
```

откомпилируется правильно:

```
}
```

Однако более подходящее место для директивы связывания – заголовочный файл, где находится объявление функции, описывающее ее интерфейс.

Как сделать C++ функцию доступной для программы на C? Директива `extern "C"`

```
// функция calc() может быть вызвана из программы на C
```

поможет и в этом:

```
extern "C" double calc( double dparm ) { /* ... */ }
```

Если в одном файле имеется несколько объявлений функции, то директива связывания может быть указана при каждом из них или только при первом – в этом случае она

```
// ---- myMath.h ----
extern "C" double calc( double );

// ---- myMath.C ----
// объявление calc() в myMath.h
#include "myMath.h"

// определение функции extern "C" calc()
// функция calc() может быть вызвана из программы на C
```

распространяется и на все последующие объявления. Например:

```
double calc( double dparm ) { // ... }
```

В данном разделе мы видели примеры директивы связывания `extern "C"` только для языка C. Это единственный внешний язык, поддержку которого гарантирует стандарт C++. Конкретная реализация может поддерживать связь и с другими языками. Например, `extern "Ada"` для функций, написанных на языке Ada; `extern "FORTRAN"` для языка FORTRAN и т.д. Мы описали один из случаев использования ключевого слова `extern` в C++. В разделе 8.2 мы покажем, что это слово имеет и другое назначение в объявлениях функций и объектов.

Упражнение 7.14

`exit()`, `printf()`, `malloc()`, `strcpy()` и `strlen()` являются функциями из библиотеки C. Модифицируйте приведенную ниже C-программу так, чтобы она компилировалась и связывалась в C++.

```
const char *str = "hello";

void *malloc( int );
char *strcpy( char *, const char * );
int printf( const char *, ... );
int exit( int );
int strlen( const char * );

int main()
{   /* программа на языке C */

    char* s = malloc( strlen(str)+l );
    strcpy( s, str );
    printf( "%s, world\n", s );
    exit( 0 );

}
```

7.8. Функция main(): разбор параметров командной строки

При запуске программы мы, как правило, передаем ей информацию в командной строке. Например, можно написать

```
prog -d -o of lie data0
```

Фактические параметры являются аргументами функции `main()` и могут быть получены из массива С-строк с именем `argv`; мы покажем, как их использовать.

Во всех предыдущих примерах определение `main()` содержало пустой список:

```
int main() { ... }
```

Развернутая сигнатура `main()` позволяет получить доступ к параметрам, которые были заданы пользователем в командной строке:

```
int main( int argc, char *argv[] ){...}
```

`argc` содержит их количество, а `argv` – С-строки, представляющие собой отдельные значения (в командной строке они разделяются пробелами). Скажем, при запуске команды

```
prog -d -o ofile data0
```

`argc` получает значение 5, а `argv` включает следующие строки:

```
argv[ 0 ] = "prog";
argv[ 1 ] = "-d";
argv[ 2 ] = "-o";
argv[ 3 ] = "ofile";
argv[ 4 ] = "data0";
```

В argv[0] всегда входит имя команды (программы). Элементы с индексами от 1 до argc-1 служат параметрами.

Посмотрим, как можно извлечь и использовать значения, помещенные в argv. Пусть

```
prog [-d] [-h] [-v]
     [-o output_file] [-l limit_value]
     file_name
```

программа из нашего примера вызывается таким образом:

```
     [ file_name [file_name [ ... ]]]
```

Параметры в квадратных скобках являются необязательными. Вот, например, запуск программы с их минимальным количеством – одним лишь именем файла:

```
prog chap1.doc
```

```
prog -l 1024 -o chap1-2.out chapl.doc chap2.doc
prog d chap3.doc
```

Но можно запускать и так:

```
prog -l 512 -d chap4.doc
```

При разборе параметров командной строки выполняются следующие основные шаги:

1. По очереди извлечь каждый параметр из argv. Мы используем для этого цикл for с

```
for ( int ix = 1; ix < argc; ++ix ) {
    char *pchar = argv[ ix ];
    // ...
```

   начальным индексом 1 (пропуская, таким образом, имя программы):

```
}
```

2. Определить тип параметра. Если строка начинается с дефиса (-), это одна из опций { h, d, v, l, o}. В противном случае это может быть либо значение, ассоциированное с опцией (максимальный размер для -l, имя выходного файла для -o), либо имя входного файла. Чтобы определить, начинается ли строка с дефиса, используем

```
switch ( pchar[ 0 ] ) {
   case '-': {
     //  -h, -d, -v, -l, -o
   }

   default: {
     // обработаем максимальный размер для опции -1
     //            имя выходного файла для       -o
     //            имена входных файлов ...
   }
```

   инструкцию switch:

```
}
```

Реализуем обработку двух случаев пункта 2.

Если строка начинается с дефиса, мы используем switch по следующему символу для

```
case '-': {
  switch( pchar[ 1 ] )
  {
    case 'd':
      // обработка опции debug
      break;

    case 'v':
      // обработка опции version
      break;

    case 'h':
      // обработка опции help
      break;

    case 'o':
      // приготовимся обработать выходной файл
      break;

    case 'l':
      // приготовимся обработать макс.размер
      break;

    default:
      // неопознанная опция:
      // сообщить об ошибке и завершить выполнение
  }
```

определения конкретной опции. Вот общая схема этой части программы:

```
}
```

Опция -d задает необходимость отладки. Ее обработка заключается в присваивании переменной с объявлением

```
bool debug_on = false;
```

```
case 'd':
  debug_on = true;
```

значения true:

```
break;
```

```
if ( debug_on )
```

В нашу программу может входить код следующего вида:

```
    display_state_elements( obj );
```

```
case 'v':
  cout << program_name << "::"
       << program_version << endl;
```

Опция `-v` выводит номер версии программы и завершает исполнение:

```
  return 0;
```

Опция `-h` запрашивает информацию о синтаксисе запуска и завершает исполнение.

```
case 'h':
  // break не нужен: usage() вызывает exit()
```

Вывод сообщения и выход из программы выполняется функцией `usage()`:

```
  usage();
```

Опция `-o` сигнализирует о том, что следующая строка содержит имя выходного файла. Аналогично опция `-l` говорит, что за ней указан максимальный размер. Как нам обработать эти ситуации?

Если в строке параметра нет дефиса, возможны три варианта: параметр содержит имя выходного файла, максимальный размер или имя входного файла. Чтобы различать эти

```
// если ofi1e_on==true,
// следующий параметр - имя выходного файла
bool ofi1e_on = false;

// если ofi1e_on==true,
// следующий параметр - максимальный размер
```

случаи, присвоим `true` переменным, отражающим внутреннее состояние:

```
bool limit_on = false;
```

```
case 'l':
  limit_on = true;
  break;

case 'o':
  ofile_on = true;
```

Вот обработка опций `-l` и `-o` в нашей инструкции `switch`:

```
  break;
```

Встретив строку, не начинающуюся с дефиса, мы с помощью переменных состояния можем узнать ее содержание:

```
// обработаем максимальный размер для опции -1
//           имя выходного файла для       -o
//           имена входных файлов ...
default: {
   // ofile_on включена, если -o встречалась
   if ( ofile_on ) {
     // обработаем имя выходного файла
     // выключим ofile_on
   }
   else if ( limit_on ) { // если -l встречалась
     // обработаем максимальный размер
     // выключим limit_on
   } else {
     // обработаем имя входного файла
   }
}
```

Если аргумент является именем выходного файла, сохраним это имя и выключим

```
if ( ofile_on ) {
  ofile_on = false;
  ofile = pchar;
```

ofile_on:

```
}
```

Если аргумент задает максимальный размер, мы должны преобразовать строку встроенного типа в представляемое ею число. Сделаем это с помощью стандартной функции atoi(), которая принимает строку в качестве аргумента и возвращает int (также существует функция atof(), возвращающая double). Для использования atoi() включим заголовочный файл ctype.h. Нужно проверить, что значение максимального

```
// int limit;
else
if ( limit_on ) {
  limit_on = false;
  limit = atoi( pchar );
  if ( limit < 0 ) {
    cerr << program_name << "::"
         << program_version << " : error: "
         << "negative value for limit.\n\n";
    usage( -2 );
  }
```

размера неотрицательно и выключить limit_on:

```
}
```

Если обе переменных состояния равны false, у нас есть имя входного файла. Сохраним

```
else
```

его в векторе строк:

```
  file_names.push_back( string( pchar ));
```

При обработке параметров командной строки важен способ реакции на неверные опции. Мы решили, что задание отрицательной величины в качестве максимального размера будет фатальной ошибкой. Это приемлемо или нет в зависимости от ситуации. Также можно распознать эту ситуацию как ошибочную, выдать предупреждение и использовать ноль или какое-либо другое значение по умолчанию.

Слабость нашей реализации становится понятной, если пользователь небрежно относится к пробелам, разделяющим параметры. Скажем, ни одна из следующих двух строк не

```
prog - d dataOl
```

будет обработана:

```
prog -oout_file dataOl
```

(Оба случая мы оставим для упражнений в конце раздела.)

Вот полный текст нашей программы. (Мы добавили инструкции печати для трассировки выполнения.)

```
#include <iostream>

#include <string>
#include <vector>

#include <ctype.h>

const char *const program_name = "comline";
const char *const program_version = "version 0.01 (08/07/97)";

inline void usage( int exit_value = 0 )
{
  // печатает отформатированное сообщение о порядке вызова
  // и завершает программу с кодом exit_value ...

  cerr << "порядок вызова:\n"
       << program_name << " "
       << "[-d] [-h] [-v] \n\t"
       << "[-o output_file] [-l limit] \n\t"
       << "file_name\n\t[file_name [file_name [ ... ]]]\n\n"
       << "где [] указывает на необязательность опции:\n\n\t"
       << "-h: справка.\n\t\t"
       << "печать этого сообщения и выход\n\n\t"
       << "-v: версия.\n\t\t"
       << "печать информации о версии программы и выход\n\n\t"
       << "-d: отладка.\n\t\t включает отладочную печать\n\n\t"
       << "-l limit\n\t\t"
       << "limit должен быть неотрицательным целым числом\n\n\t"
       << "-o ofile\n\t\t"
       << "файл, в который выводится результат\n\t\t"
       << "по умолчанию результат записывается на стандартный вывод\n\n"
       << "file_name\n\t\t"
       << "имя подлежащего обработке файла\n\t\t"
       << "должно быть задано хотя бы одно имя --\n\t\t"
       << "но максимальное число не ограничено\n\n"
       << "примеры:\n\t\t"
       << "$command chapter7.doc\n\t\t"
       << "$command -d -l 1024 -o test_7_8 "
       << "chapter7.doc chapter8.doc\n\n";

  exit( exit_value );
}

int main( int argc, char* argv[] )
{
  bool debug_on = false;
  bool ofile_on = false;
  bool limit_on = false;
  int limit = -1;
  string ofile;
  vector<string> file_names;

  cout << "демонстрация обработки параметров в командной строке:\n"
       << "argc: " << argc << endl;

  for ( int ix = 1; ix < argc; ++ix )
  {
    cout << "argv[ " << ix << " ]: "
         << argv[ ix ] << endl;

    char *pchar = argv[ ix ];
    switch ( pchar[ 0 ] )
    {
      case '-':
      {
        cout << "встретился \'-\'\n";
        switch( pchar[ 1 ] )
        {
          case 'd':
            cout << "встретилась -d: "
                 << "отладочная печать включена\n";

            debug_on = true;
            break;

          case 'v':
            cout << "встретилась -v: "
```

```
}


a.out -d -l 1024 -o test_7_8 chapter7.doc  chapters.doc
```

Вот трассировка обработки параметров командной строки:

```
демонстрация обработки параметров в командной строке:
argc: 8
argv[ 1 ]: -d
встретился '-'
встретилась -d: отладочная печать включена
argv[ 2 ]: -l
встретился '-'
встретилась -l: ограничение ресурса
argv[ 3 ]: 1024
default: параметр без дефиса: 1024
argv[ 4 ]: -o
встретился '-'
встретилась -o: выходной файл
argv[ 5 ]: test_7_8
default: параметр без дефиса: test_7_8
argv[ 6 ]: chapter7.doc
default: параметр без дефиса: chapter7.doc
argv[ 7 ]: chapter8.doc
default: параметр без дефиса: chapter8.doc
Заданное пользователем значение limit: 1024
Заданный пользователем выходной файл: test_7_8
Файлы, подлежащий(е) обработке:
        chapter7.doc
        chapter8.doc
```

7.8.1. Класс для обработки параметров командной строки

Чтобы не перегружать функцию `main()` деталями, касающимися обработки параметров командной строки, лучше отделить этот фрагмент. Можно написать для этого функцию.

```
extern int parse_options( int arg_count, char *arg_vector );

int main( int argc, char *argv[] ) {
    // ...
    int option_status;
    option_status = parse_options( argc, argv );
    // ...
```

Например:

```
}
```

Как вернуть несколько значений? Обычно для этого используются глобальные объекты, которые не передаются ни в функцию для их обработки, ни обратно. Альтернативной стратегией является инкапсуляция обработки параметров командной строки в класс.

Данные-члены класса представляют собой параметры, заданные пользователем в командной строке. Набор открытых встроенных функций-членов позволяет получать их значения. Конструктор инициализирует параметры значениями по умолчанию. Функция-член получает `argc` и `argv` в качестве аргументов и обрабатывает их:

```
#include <vector>
#include <string>

class CommandOpt {
public:
    CommandOpt() : _limit( -1 ), _debug_on( false ) {}
    int parse_options( int argc, char *argv[] );

    string out_file() { return _out_file; }
    bool   debug_on() { return _debug_on; }
    int    files()    { return _file_names.size(); }

    string& operator[]( int ix );

private:
    inline void usage( int exit_value = 0 );

    bool _debug_on;
    int _limit;
    string _out_file;
    vector<string> _file_names;

    static const char *const program_name;
    static const char *const program_version;

};
```

```
#include "CommandOpt.h"

int main( int argc, char "argv[] ) {
   // ...
   CommandOpt com_opt;
   int option_status;
   opttion_status = com_opt. parse_options (argc, argv);
   // ...
```

Так выглядит модифицированная функция `main()`:[18]

```
}
```

Упражнение 7.15

Добавьте обработку опций `-t` (включение таймера) и `-b` (задание размера буфера `bufsize`). Не забудьте обновить `usage()`. Например:

```
prog -t -b 512 data0
```

Упражнение 7.16

Наша реализация не обрабатывает случая, когда между опцией и ассоциированным с ней значением нет пробела. Модифицируйте программу для поддержки такой обработки.

Упражнение 7.17

Наша реализация не может различить лишний пробел между дефисом и опцией:

---

18 Полный текст реализации класса CommandOpt можно найти на Web-сайте издательства Addison-Wesley.

```
prog - d data0
```

Модифицируйте программу так, чтобы она распознавала подобную ошибку и сообщала о ней.

Упражнение 7.18

В нашей программе не предусмотрен случай, когда опции `-l` или `-o` задаются несколько раз. Реализуйте такую возможность. Какова должна быть стратегия при разрешении конфликта?

Упражнение 7.19

В нашей реализации задание неизвестной опции приводит к фатальной ошибке. Как вы думаете, это оправдано? Предложите другое поведение.

Упражнение 7.20

Добавьте поддержку опций, начинающихся со знака плюс (+), обеспечив обработку `+s` и `+pt`, а также `+sp` и `+ps`. Предположим, что `+s` включает строгую проверку синтаксиса, а `+p` допускает использование устаревших конструкций. Например:

```
prog +s +p -d -b 1024 data0
```

7.9. Указатели на функции

Предположим, что нам нужно написать функцию сортировки, вызов которой выглядит так:

```
sort( start, end, compare );
```

где `start` и `end` являются указателями на элементы массива строк. Функция `sort()` сортирует элементы между `start` и `end`, а аргумент `compare` задает операцию сравнения двух строк этого массива.

Какую реализацию выбрать для `compare`? Мы можем сортировать строки лексикографически, т.е. в том порядке, в котором слова располагаются в словаре, или по длине – более короткие идут раньше более длинных. Нам нужен механизм для задания альтернативных операций сравнения.

(Заметим, что в главе 12 описан алгоритм `sort()` и другие обобщенные алгоритмы из стандартной библиотеки C++. В этом разделе мы покажем свою собственную версию `sort()` как пример употребления указателей на функции. Наша функция будет упрощенным вариантом стандартного алгоритма.)

Один из способов удовлетворить наши потребности – использовать в качестве третьего аргумента `compare` указатель на функцию, применяемую для сравнения.

Для того чтобы упростить использование функции `sort()`, не жертвуя гибкостью, можно задать операцию сравнению по умолчанию, подходящую для большинства случаев. Предположим, что чаще всего нам требуется лексикографическая сортировка, поэтому в качестве такой операции возьмем функцию `compare()` для строк (эта функция впервые встретилась в разделе 6.10).

7.9.1. Тип указателя на функцию

Как объявить указатель на функцию? Как выглядит формальный параметр, когда фактическим аргументом является такой указатель? Вот определение функции

```
#include <string>
int lexicoCompare( const string &sl, const string &s2 ) {
    return sl.compare(s2);
}
```

lexicoCompare(), которая сравнивает две строки лексикографически:

Если все символы строк s1 и s2 равны, lexicoCompare() вернет 0, в противном случае – отрицательное число, если s1 меньше чем s2, и положительное, если s1 больше s2.

Имя функции не входит в ее сигнатуру – она определяется только типом возвращаемого значения и списком параметров. Указатель на lexicoCompare() должен адресовать

```
int *pf( const string &, const string & ) ;
```

функцию с той же сигнатурой. Попробуем написать так:

```
// нет, не совсем так
```

Эта инструкция почти правильна. Проблема в том, что компилятор интерпретирует ее как объявление функции с именем pf, которая возвращает указатель типа int*. Список параметров правилен, но тип возвращаемого значения не тот. Оператор разыменования (*) ассоциируется с данным типом (int в нашем случае), а не с pf. Чтобы исправить

```
int (*pf)( const string &, const string & ) ;
```

положение, нужно использовать скобки:

```
// правильно
```

pf объявлен как указатель на функцию с двумя параметрами, возвращающую значение типа int, т.е. такую, как lexicoCompare().

pf способен адресовать и приведенную ниже функцию, поскольку ее сигнатура совпадает с типом lexicoCompare():

```
int sizeCompare( const string &sl, const string &s2 );
```

```
int calc( int , int );
```

Функции calc() и gcd()другого типа, поэтому pf не может указывать на них:

```
int gcd( int , int );
```

Указатель, который адресует эти две функции, определяется так:

```
int (*pfi)( int, int );
```

Многоточие является частью сигнатуры функции. Если у двух функций списки параметров отличаются только тем, что в конце одного из них стоит многоточие, то

```
int printf( const char*, ... );
int strlen( const char* );

int (*pfce)( const char*, ... ); // может указывать на printf()
```

считается, что функции различны. Таковы же и типы указателей.

```
int (*pfc)( const char* );       // может указывать на strlen()
```

Типов функций столько, сколько комбинаций типов возвращаемых значений и списков параметров.

7.9.2. Инициализация и присваивание

Вспомним, что имя массива без указания индекса элемента интерпретируется как адрес первого элемента. Аналогично имя функции без следующих за ним скобок интерпретируется как указатель на функцию. Например, при вычислении выражения

```
lexicoCompare;
```

получается указатель типа

```
int (*)( const string &, const string & );
```

Применение оператора взятия адреса к имени функции также дает указатель того же типа, например `lexicoCompare` и `&lexicoCompare`. Указатель на функцию

```
int (*pfi)( const string &, const string & ) = lexicoCompare;
```

инициализируется следующим образом:

```
int (*pfi2)( const string &, const string & ) = &lexicoCompare;
```

```
pfi = lexicoCompare;
```

Ему можно присвоить значение:

```
pfi2 = pfi;
```

Инициализация и присваивание корректны только тогда, когда список параметров и тип значения, которое возвращает функция, адресованная указателем в левой части операции присваивания, в точности соответствуют списку параметров и типу значения, возвращаемого функцией или указателем в правой части. В противном случае выдается сообщение об ошибке компиляции. Никаких неявных преобразований типов для указателей на функции не производится. Например:

```
int calc( int, int );
int (*pfi2s)( const string &, const string & ) = 0;
int (*pfi2i)( int, int ) = 0;
int main() {
    pfi2i = calc;  // правильно
    pri2s = calc;  // ошибка: несовпадение типов
    pfi2s = pfi2i; // ошибка: несовпадение типов
    return 0;
}
```

Такой указатель можно инициализировать нулем или присвоить ему нулевое значение, в этом случае он не адресует функцию.

7.9.3. Вызов

Указатель на функцию применяется для вызова функции, которую он адресует. Включать оператор разыменования при этом необязательно. И прямой вызов функции по имени, и

```
#include <iostream>

int min( int*, int );
int (*pf)( int*, int ) = min;

const int iaSize = 5;
int ia[ iaSize ] = { 7, 4, 9, 2, 5 };

int main() {
    cout << "Прямой вызов: min: "
         << min( ia, iaSize ) << endl;

    cout << "Косвенный вызов: min: "
         << pf( ia, iaSize ) << endl;

    return 0;
}

int min( int* ia, int sz ) {
    int minVal = ia[ 0 ];
    for ( int ix = 1; ix < sz; ++ix )
        if ( minVal > ia[ ix ] )
            minVal = ia[ ix ];
    return minVal;
```

косвенный вызов по указателю записываются одинаково:

```
}
```

Вызов

```
pf( ia, iaSize );
```

может быть записан также и с использованием явного синтаксиса указателя:

```
(*pf)( ia, iaSize );
```

Результат в обоих случаях одинаковый, но вторая форма говорит читателю, что вызов осуществляется через указатель на функцию.

Конечно, если такой указатель имеет нулевое значение, то любая форма вызова приведет к ошибке во время выполнения. Использовать можно только те указатели, которые адресуют какую-либо функцию или были проинициализированы таким значением.

7.9.4. Массивы указателей на функции

Можно объявить массив указателей на функции. Например:

```
int (*testCases[10])();
```

testCases – это массив из десяти элементов, каждый из которых является указателем на функцию, возвращающую значение типа int и не имеющую параметров.

Подобные объявления трудно читать, поскольку не сразу видно, с какой частью ассоциируется тип функции.

В этом случае помогает использование имен, определенных с помощью директивы

```
// typedef делает объявление более понятным
typedef int (*PFV)(); // typedef для указателя на функцию
```

typedef:

```
PFV testCases[10];
```

Данное объявление эквивалентно предыдущему.

Вызов функций, адресуемых элементами массива testCases, выглядит следующим

```
const int size = 10;
PFV testCases[size];
int testResults[size];

void runtests() {
    for ( int i = 0; i < size; ++i )
        // вызов через элемент массива
        testResults[ i ] = testCases[ i ]();
```

образом:

```
}
```

Массив указателей на функции может быть инициализирован списком, каждый элемент которого является функцией. Например:

```
int lexicoCompare( const string &, const string & );
int sizeCompare( const string &, const string & );

typedef int ( *PFI2S )( const string &, const string & );
PFI2S compareFuncs[2] =
{
    lexicoCompare,
    sizeCompare
};
```

Можно объявить и указатель на compareFuncs, его типом будет “указатель на массив указателей на функции”:

```
PFI2S (*pfCompare)[2] = compareFuncs;
```

Это объявление раскладывается на составные части следующим образом:

```
(*pfCompare)
```

Оператор разыменования говорит, что pfCompare является указателем. [2] сообщает о количестве элементов массива:

```
 (*pfCompare) [2]
```

PFI2S – имя, определенное с помощью директивы typedef, называет тип элементов. Это “указатель на функцию, возвращающую int и имеющую два параметра типа const string &”. Тип элемента массива тот же, что и выражения &lexicoCompare.

Такой тип имеет и первый элемент массива compareFuncs, который может быть получен

```
compareFunc[ 0 ];
```

с помощью любого из выражений:

```
(*pfCompare)[ 0 ];
```

Чтобы вызвать функцию lexicoCompare через pfCompare, нужно написать одну из

```
// эквивалентные вызовы
pfCompare [ 0 ]( string1, string2 );     // сокращенная форма
```

следующих инструкций:

```
((*pfCompare)[ 0 ])( string1, string2 ); // явная форма
```

7.9.5. Параметры и тип возврата

Вернемся к задаче, сформулированной в начале данного раздела. Как использовать указатели на функции для сортировки элементов? Мы можем передать в алгоритм сортировки указатель на функцию, которая выполняет сравнение:

```
int sort( string*, string*,
          int (*)( const string &, const string & ) );
```

И в этом случае директива typedef помогает сделать объявление sort() более

```
// Использование директивы typedef делает
// объявление sort() более понятным
typedef int ( *PFI2S )( const string &, const string & );
```

понятным:

```
int sort( string*, string*, PFI2S );
```

Поскольку в большинстве случаев употребляется функция lexicoCompare, можно

```
// значение по умолчанию для третьего параметра
int lexicoCompare( const string &, const string & );
```

использовать значение параметра по умолчанию:

```
int sort( string*, string*, PFI2S = lexicoCompare );
```

```
1 void sort( string *sl, string *s2,
2            PFI2S compare = lexicoCompare )
3 {
4     // условие окончания рекурсии
5     if ( si < s2 ) {
6         string elem = *s1;
7         string *1ow = s1;
8         string *high = s2 + 1;
9
10        for (;;) {
11            while ( compare ( *++1ow, elem ) < 0 && low < s2) ;
12            while ( compare( elem, *--high ) < 0 && high > s1)
14            if ( low < high )
15                1ow->swap(*high);
16            else break;
17        } // end, for(;;)
18
19        s1->swap(*high);
20        sort( s1, high - 1 );
21        sort( high +1, s2 );
22    } // end, if ( si < s2 )
```

Определение sort() выглядит следующим образом:

```
23 }
```

sort() реализует алгоритм *быстрой сортировки Хоара* (C.A.R.Hoare). Рассмотрим ее определение детально. Она сортирует элементы массива от s1 до s2. Это рекурсивная функция, которая вызывает сама себя для последовательно уменьшающихся подмассивов. Рекурсия окончится тогда, когда s1 и s2 укажут на один и тот же элемент или s1 будет располагаться после s2 (строка 5).

`elem` (строка 6) является *разделяющим элементом*. Все элементы, меньшие чем `elem`, перемещаются влево от него, а большие – вправо. Теперь массив разбит на две части. `sort()` рекурсивно вызывается для каждой из них (строки 20-21).

Цикл `for(;;)` проводит разделение (строки 10-17). На каждой итерации цикла индекс `low` увеличивается до первого элемента, большего или равного `elem` (строка 11). Аналогично `high` уменьшается до последнего элемента, меньшего или равного `elem` (строка 12). Когда `low` становится равным или большим `high`, мы выходим из цикла, в противном случае нужно поменять местами значения элементов и начать новую итерацию (строки 14-16). Хотя элементы разделены, `elem` все еще остается первым в массиве. `swap()` в строке 19 ставит его на место до рекурсивного вызова `sort()` для двух частей массива.

Сравнение производится вызовом функции, на которую указывает `compare` (строки 11-12). Чтобы поменять элементы массива местами, используется операция `swap()` с аргументами типа `string`, представленная в разделе 6.11.

```
#include <iostream>
#include <string>

// это должно бы находиться в заголовочном файле
int lexicoCompare( const string &, const string & );
int sizeCompare( const string &, const string & );
typedef int (*PFI)( const string &, const string & );
void sort( string *, string *, PFI=lexicoCompare );

string as[10] = { "a", "light", "drizzle", "was", "falling",
                  "when", "they", "left", "the", "museum" };

int main() {
    // вызов sort() с значением по умолчанию параметра compare
    sort( as, as + sizeof(as)/sizeof(as[0]) - 1 );

   // выводим результат сортировки
   for ( int i = 0; i < sizeof(as)/sizeof(as[0]); ++i )
       cout << as[ i ].c_str() << "\n\t";
```

Вот как выглядит `main()`, в которой применяется наша функция сортировки:

```
}
```

Результат работы программы:

```
"a"
"drizzle"
"falling"
"left"
"light"
"museum"
"the"
"they"
"was"
"when"
```

Параметр функции автоматически приводится к типу указателя на функцию:

```
// typedef представляет собой тип функции
typedef int functype( const string &, const string & );

void sort( string *, string *, functype );


void sort( string *, string *,
```

`sort()` рассматривается компилятором как объявленная в виде

```
           int (*)( const string &, const string & ) );
```

Два этих объявления `sort()` эквивалентны.

Заметим, что, помимо использования в качестве параметра, указатель на функцию может быть еще и типом возвращаемого значения. Например:

```
int (*ff( int ))( int*, int );
```

`ff()` объявляется как функция, имеющая один параметр типа `int` и возвращающая указатель на функцию типа

```
int (*)( int*, int );
```

И здесь использование директивы `typedef` делает объявление понятнее. Объявив `PF` с

```
// Использование директивы typedef делает
// объявления более понятными
typedef int (*PF)( int*, int );
```

помощью `typedef`, мы видим, что `ff()` возвращает указатель на функцию:

```
PF ff( int );
```

Типом возвращаемого значения функции не может быть тип функции. В этом случае

```
// typedef представляет собой тип функции
typedef int func( int*, int );
```

выдается ошибка компиляции. Например, нельзя объявить `ff()` таким образом:

```
func ff( int ); // ошибка: тип возврата ff() - функция
```

7.9.6. Указатели на функции, объявленные как extern "C"

Можно объявлять указатели на функции, написанные на других языках программирования. Это делается с помощью директивы связывания. Например, указатель `pf` ссылается на C-функцию:

```
extern "C" void (*pf)(int);
```

```
extern "C" void exit(int);

// pf ссылается на C-функцию exit()
extern "C" void (*pf)(int) = exit;
int main() {
    // ...
    // вызов C-функции, а именно exit()
    (*pf)(99);
```

Через pf вызывается функция, написанная на языке C.

```
}
```

Вспомним, что присваивание и инициализация указателя на функцию возможны лишь тогда, когда тип в левой части оператора присваивания в точности соответствует типу в правой его части. Следовательно, указатель на C-функцию не может адресовать функцию C++ (и инициализация его таким адресом не допускается), и наоборот. Подобная попытка

```
void (*pfl)(int);
extern "C" void (*pf2)(int);
int main() {
    pfl = pf2; // ошибка: pfl и pf2 имеют разные типы
    // ...
```

вызывает ошибку компиляции:

```
}
```

Отметим, что в некоторых реализациях C++ характеристики указателей на функции C и C++ одинаковы. Отдельные компиляторы могут допустить подобное присваивание, рассматривая это как расширение языка.

Если директива связывания применяется к объявлению, она затрагивает все функции, участвующие в данном объявлении.

В следующем примере параметр pfParm также служит указателем на C-функцию. Директива связывания применяется к объявлению функции, к которой этот параметр

```
// pfParm - указатель на C-функцию
```

относится:

```
extern "C" void f1( void(*pfParm)(int) );
```

Следовательно, f1() является C-функцией с одним параметром – указателем на C-функцию. Значит, передаваемый ей аргумент должен быть либо такой же функцией, либо указателем на нее, поскольку считается, что указатели на функции, написанные на разных языках, имеют разные типы. (Снова заметим, что в тех реализациях C++, где указатели на функции C и C++ имеют одинаковые характеристики, компилятор может поддерживать расширение языка, позволяющее не различать эти два типа указателей.)

Коль скоро директива связывания относится ко всем функциям в объявлении, то как же объявить функцию C++, имеющую в качестве параметра указатель на C-функцию? С помощью директивы typedef. Например:

```
// FC представляет собой тип:
// С-функция с параметром типа int, не возвращающая никакого значения
extern "C" typedef void FC( int );

// f2() - C++ функция с параметром -
// указателем на С-функцию

void f2( FC *pfParm );
```

Упражнение 7.21

В разделе 7.5 приводится определение функции `factorial()`. Напишите объявление указателя на нее. Вызовите функцию через этот указатель для вычисления факториала 11.

Упражнение 7.22

```
(a) int (*mpf)(vector<int>&);
(b) void (*apf[20])(doub1e);
```

Каковы типы следующих объявлений:

```
(c) void (*(*papf)[2])(int);
```

Как сделать эти объявления более понятными, используя директивы `typedef`?

Упражнение 7.23

```
double abs(double);
double sin(double);
double cos(double);
```

Вот функции из библиотеки С, определенные в заголовочном файле `<cmath>`:

```
double sqrt(double);
```

Как бы вы объявили массив указателей на С-функции и инициализировали его этими четырьмя функциями? Напишите `main()`, которая вызывает `sqrt()` с аргументом `97.9` через элемент массива.

Упражнение 7.24

Вернемся к примеру `sort()`. Напишите определение функции

```
int sizeCompare( const string &, const string & );
```

Если передаваемые в качестве параметров строки имеют одинаковую длину, то `sizeCompare()` возвращает `0`; если первая строка короче второй, то отрицательное число, а если длиннее, то положительное. Напоминаем, что длина строки возвращается операцией `size()` класса `string`. Измените `main()` для вызова `sort()`, передав в качестве третьего аргумента указатель на `sizeCompare()`.

8. Область видимости и время жизни

В этой главе обсуждаются два важных вопроса, касающиеся объявлений в C++. Где употребляется объявленное имя? Когда можно безопасно использовать объект или вызывать функцию, т.е. каково время жизни сущности в программе? Для ответа на первый вопрос мы введем понятие областей видимости и покажем, как они ограничивают применение имен в исходном файле программы. Мы рассмотрим разные типы таких областей: глобальную и локальную, а также более сложное понятие областей видимости пространств имен, которое появится в конце главы. Отвечая на второй вопрос, мы опишем, как объявления вводят глобальные объекты и функции (сущности, “живущие” в течение всего времени работы программы), локальные (“живущие” на определенном отрезке выполнения) и динамически размещаемые объекты (временем жизни которых управляет программист). Мы также исследуем свойства времени выполнения, характерные для этих объектов и функций.

8.1. Область видимости

Каждое имя в C++ программе должно относиться к уникальной сущности (объекту, функции, типу или шаблону). Это не значит, что оно встречается только один раз во всей программе: его можно повторно использовать для обозначения другой сущности, если только есть некоторый *контекст*, помогающий различить разные значения одного и того же имени. Контекстом, служащим для такого различения, служит *область видимости*. В C++ поддерживается три их типа: *локальная* область видимости, область видимости *пространства имен* и область видимости *класса*.

Локальная область – это часть исходного текста программы, содержащаяся в определении функции (или в блоке). Любая функция имеет собственную такую часть, и каждая составная инструкция (или блок) внутри функции также представляет собой отдельную локальную область.

Область видимости пространства имен – часть исходного текста программы, не содержащаяся внутри объявления или определения функции или определения класса. Самая внешняя часть называется глобальной областью видимости или глобальной областью видимости пространства имен.

Объекты, функции, типы и шаблоны могут быть определены в глобальной области видимости. Программисту разрешено задать *пользовательские* пространства имен, заключенные внутри глобальной области с помощью *определения пространства имен*. Каждое такое пространство является отдельной областью видимости. Пользовательское пространство, как и глобальное, может содержать объявления и определения объектов, функций, типов и шаблонов, а также вложенные пользовательские пространства имен. (Они рассматриваются в разделах 8.5 и 8.6.)

Каждое определение класса представляет собой *отдельную область видимости класса*. (О таких областях мы расскажем в главе 13.)

Имя может обозначать различные сущности в зависимости от области видимости. В следующем фрагменте программы имя `s1` относится к четырем разным сущностям:

```
#include <iostream>
#include <string>

// сравниваем s1 и s2 лексикографически
int lexicoCompare( const string &sl, const string &s2 ) { ... }

// сравниваем длины s1 и s2
int sizeCompare( const string &sl, const string &s2 ) { ... }

typedef int ( PFI)( const string &, const string & );
// сортируем массив строк
void sort( string *s1, string *s2, PFI compare =lexicoCompare )
    { ... }

string sl[10] = { "a", "light", "drizzle", "was", "falling",
                  "when", "they", "left", "the", "school" };
int main()
{
    // вызов sort() со значением по умолчанию параметра compare
    // s1 - глобальный массив
    sort( s1, s1 + sizeof(s1)/sizeof(s1[0]) - 1 );

    // выводим результат сортировки
    for ( int i = 0; i < sizeof(s1) / sizeof(s1[0]); ++i )
         cout << s1[ i ].c_str() << "\n\t";

}
```

Поскольку определения функций `lexicoCompare()`, `sizeCompare()` и `sort()` представляют собой различные области видимости и все они отличны от глобальной, в каждой из этих областей можно завести переменную с именем `s1`.

Имя, введенное с помощью объявления, можно использовать от точки объявления до конца области видимости (включая вложенные области). Так, имя `s1` параметра функции `lexicoCompare()` разрешается употреблять до конца ее области видимости, то есть до конца ее определения.

Имя глобального массива `s1` видимо с точки его объявления до конца исходного файла, включая вложенные области, такие, как определение функции `main()`.

В общем случае имя должно обозначать одну сущность внутри одной области видимости. Если в предыдущем примере после объявления массива `s1` добавить следующую строку, компилятор выдаст сообщение об ошибке:

```
void s1(); // ошибка: повторное объявление s1
```

Перегруженные функции являются исключением из правила: можно завести несколько одноименных функций в одной области видимости, если они отличаются списком параметров. (Перегруженные функции рассматриваются в главе 9.)

В C++ имя должно быть объявлено до момента его первого использования в выражении. В противном случае компилятор выдаст сообщение об ошибке. Процесс сопоставления имени, используемого в выражении, с его объявлением называется *разрешением*. С помощью этого процесса имя получает конкретный смысл. Разрешение имени зависит от способа его употребления и от его области видимости. Мы рассмотрим этот процесс в различных контекстах. (В следующем подразделе описывается разрешение имен в локальной области видимости; в разделе 10.9 – разрешение в шаблонах функций; в конце главы 13 – в области видимости классов, а в разделе 16.12 – в шаблонах классов.)

Области видимости и разрешение имен – понятия времени компиляции. Они применимы к отдельным частям текста программы. Компилятор интерпретирует текст программы согласно правилам областей видимости и правилам разрешения имен.

8.1.1. Локальная область видимости

Локальная область видимости – это часть исходного текста программы, содержащаяся в определении функции (или блоке внутри тела функции). Все функции имеют свои локальные области видимости. Каждая составная инструкция (или блок) внутри функции также представляет собой отдельную локальную область. Такие области могут быть вложенными. Например, следующее определение функции содержит два их уровня

```
const int notFound = -1; // глобальная область видимости
int binSearch( const vector<int> &vec, int val )
{ // локальная область видимости: уровень #1
    int low = 0;
    int high = vec.size() - 1;

    while ( low <= high )
    { // локальная область видимости: уровень #2
        int mid = ( low + high ) / 2;
        if ( val < vec[ mid ] )
            high = mid - 1;
        else low = mid + 1;
    }
    return notFound; // локальная область видимости: уровень #1
```

(функция выполняет двоичный поиск в отсортированном векторе целых чисел):

```
}
```

Первая локальная область видимости – тело функции `binSearch()`. В ней объявлены параметры функции `vec` и `val`, а также переменные `low` и `high`. Цикл `while` внутри функции задает вложенную локальную область, в которой определена одна переменная `mid`. Параметры `vec` и `val` и переменные `low` и `high` видны во вложенной области. Глобальная область видимости включает в себя обе локальных. В ней определена одна целая константа `notFound`.

Имена параметров функции `vec` и `val` принадлежат к первой локальной области видимости тела функции, и в ней использовать те же имена для других сущностей нельзя.

```
int binSearch( const vector<int> &vec, int val )
{ // локальная область видимости: уровень #1
    int val; // ошибка: неверное переопределение val
```

Например:

```
   // ...
```

Имена параметров употребляются как внутри тела функции `binSearch()`, так и внутри вложенной области видимости цикла `while`. Параметры `vec` и `val` недоступны вне тела функции `binSearch()`.

Разрешение имени в локальной области видимости происходит следующим образом: просматривается та область, где оно встретилось. Если объявление найдено, имя разрешено. Если нет, просматривается область видимости, включающая текущую. Этот

процесс продолжается до тех пор, пока объявление не будет найдено либо не будет достигнута глобальная область видимости. Если и там имени нет, оно будет считаться ошибочным.

Из-за порядка просмотра областей видимости в процессе разрешения имен объявление из внешней области может быть *скрыто* объявлением того же имени во вложенной области. Если бы в предыдущем примере переменная `low` была объявлена в глобальной области видимости перед определением функции `binSearch()`, то использование `low` в локальной области видимости цикла `while` все равно относилось бы к локальному

```
int low;
int binSearch( const vector<int> &vec, int val )
{
    // локальное объявление low
    // скрывает глобальное объявление
    int low = 0;
    // ...
    // low - локальная переменная
    while ( low <= high )
    {//...
    }
    // ...
```

объявлению, скрывающему глобальное:

```
}
```

Для некоторых инструкций языка C++ разрешено объявлять переменные внутри управляющей части. Например, в цикле `for` переменную можно определить внутри

```
for ( int index = 0; index < vecSize; ++index )
{
    // переменная index видна только здесь
    if ( vec[ index ] == someValue )
        break;
}
// ошибка: переменная index не видна
```

инструкции инициализации:

```
if ( index != vecSize ) // элемент найден
```

Подобные переменные видны только в локальной области самого цикла `for` и вложенных в него (это верно для стандарта C++, в предыдущих версиях языка поведение было иным). Компилятор рассматривает это объявление так же, как если бы оно было записано

```
// представление компилятора
{  // невидимый блок
    int index = 0;
    for ( ; index < vecSize; ++index )
    {
      // ...
    }
```

в виде:

```
}
```

Тем самым программисту запрещается применять управляющую переменную вне локальной области видимости цикла. Если нужно проверить index, чтобы определить,

```
int index = 0;
for ( ; index < vecSize; ++index )
{
    // ...
}
// правильно: переменная index видна
```

было ли найдено значение, то данный фрагмент кода следует переписать так:

```
if ( index != vecSize ) // элемент найден
```

Поскольку переменная, объявленная в инструкции инициализации цикла for, является локальной для цикла, то же самое имя допустимо использовать аналогичным образом и в

```
void fooBar( int *ia, int sz )
{
     for (int i=0; i<sz; ++i) ... // правильно
     for (int i=0; i<sz; ++i) ... // правильно, другое i
     for (int i=0; i<sz; ++i) ... // правильно, другое i
```

других циклах, расположенных в данной локальной области видимости:

```
}
```

Аналогично переменная может быть объявлена внутри условия инструкций if и switch,

```
if ( int *pi = getValue() )
{
    // pi != 0 -- *pi можно использовать здесь
    int result = calc(*pi);
    // ...
}
else
{
    // здесь pi тоже видна
    // pi == 0
    cout << "ошибка: getValue() завершилась неудачно" << endl;
```

а также внутри условия циклов while и for. Например:

```
}
```

Переменные, определенные в условии инструкции if, как переменная pi, видны только внутри if и соответствующей части else, а также во вложенных областях. Значением условия является значение этой переменной, которое она получает в результате инициализации. Если pi равна 0 (нулевой указатель), условие ложно и выполняется ветвь else. Если pi инициализируется любым другим значением, условие истинно и выполняется ветвь if. (Инструкции if, switch, for и while рассматривались в главе 5.)

Упражнение 8.1

Найдите различные области видимости в следующем примере. Какие объявления ошибочны и почему?

```
int ix = 1024;
int ix() ;

void func( int ix, int iy ) {
    int ix = 255;

    if (int ix=0) {
       int ix = 79;
       {
           int ix = 89;
       }
    }
    else {
        int ix = 99;
    }
}
```

Упражнение 8.2

К каким объявлениям относятся различные использования переменных `ix` и `iy` в

```
int ix = 1024;

void func( int ix, int iy ) {
    ix = 100;

    for( int iy = 0; iy < 400; iy += 100 ) {
        iy += 100;
        ix = 300;
    }
    iy = 400;
}
```

следующем примере:

8.2. Глобальные объекты и функции

Объявление функции в глобальной области видимости вводит *глобальную функцию*, а объявление переменной – *глобальный объект*. Глобальный объект существует на протяжении всего времени выполнения программы. *Время жизни* глобального объекта начинается с момента запуска программы и заканчивается с ее завершением.

Для того чтобы глобальную функцию можно было вызвать или взять ее адрес, она должна иметь определение. Любой глобальный объект, используемый в программе, должен быть определен, причем только один раз. Встроенные функции могут определяться несколько раз, если только все определения совпадают. Такое требование единственности или точного совпадения получило название *правила одного определения* (ПОО). В этом разделе мы покажем, как следует вводить глобальные объекты и функции в программе, чтобы ПОО соблюдалось.

8.2.1. Объявления и определения

Как было сказано в главе 7, *объявление* функции устанавливает ее имя, а также тип возвращаемого значения и список параметров. *Определение* функции, помимо этой

информации, задает еще и тело – набор инструкций, заключенных в фигурные скобки.

```
// объявление функции calc()
// определение находится в другом файле
void calc(int);
int main()
{
    int loc1 = get(); // ошибка: get() не объявлена
    calc(loc1);       // правильно: calc() объявлена
    // ...
}
```

Функция должна быть объявлена перед вызовом. Например:

```
type_specifier object_name;
```

Определение объекта имеет две формы:

```
type_specifier object_name = initializer;
```

Вот, например, определение obj1. Здесь obj1 инициализируется значением 97:

```
int obj1 = 97;
```

Следующая инструкция задает obj2, хотя начальное значение не задано:

```
int obj2;
```

Объект, определенный в глобальной области видимости без явной инициализации, гарантированно получит нулевое значение. Таким образом, в следующих двух примерах

```
int var1 = 0;
```

и var1, и var2 будут равны нулю:

```
int var2;
```

Глобальный объект можно определить в программе только один раз. Поскольку он должен быть объявлен в исходном файле перед использованием, то для программы, состоящей из нескольких файлов, необходима возможность объявить объект, не определяя его. Как это сделать?

С помощью ключевого слова extern, аналогичного объявлению функции: оно указывает, что объект определен в другом месте – в этом же исходном файле или в другом. Например:

```
extern int i;
```

Эта инструкция “обещает”, что в программе имеется определение, подобное

```
int i;
```

extern-объявление не выделяет места под объект. Оно может встретиться несколько раз в одном и том же исходном файле или в разных файлах одной программы. Однако обычно находится в общедоступном заголовочном файле, который включается в те модули, где

```
// заголовочный файл
extern int obj1;
extern int obj2;
// исходный файл
int obj1 = 97;
```

необходимо использовать глобальный объект:

```
int obj2;
```

Объявление глобального объекта с указанием ключевого слова extern и с явной инициализацией считается определением. Под этот объект выделяется память, и другие

```
extern const double pi = 3.1416; // определение
```

определения не допускаются:

```
const double pi; // ошибка: повторное определение pi
```

Ключевое слово extern может быть указано и при объявлении функции – для явного обозначения его подразумеваемого смысла: “определено в другом месте”. Например:

```
extern void putValues( int*, int );
```

8.2.2. Сопоставление объявлений в разных файлах

Одна из проблем, вытекающих из возможности объявлять объект или функцию в разных файлах, – вероятность несоответствия объявлений или их расхождения в связи с модификацией программы. В C++ имеются средства, помогающие обнаружить такие различия.

Предположим, что в файле token.C функция addToken() определена как имеющая один параметр типа unsigned char. В файле lex.C, где эта функция вызывается, в ее

```
// ---- в файле token.C ----
int addToken( unsigned char tok ) { /* ... */ }

// ---- в файле lex.C ----
```

определении указан параметр типа char.

```
extern int addToken( char );
```

Вызов addToken() в файле lex.C вызывает ошибку во время связывания программы. Если бы такое связывание прошло успешно, можно представить дальнейшее развитие событий: скомпилированная программа была протестирована на рабочей станции Sun Sparc, а затем перенесена на IBM 390. Первый же запуск потерпел неудачу: даже самые простые тесты не проходили. Что случилось?

Вот часть объявлений набора лексем:

```
const unsigned char INLINE = 128;
const unsigned char VIRTUAL = 129;
```

```
curTok = INLINE;
// ...
```

Вызов addToken( ) выглядит так:

```
addToken( curTok );
```

Тип char реализован как знаковый в одном случае и как беззнаковый в другом. Неверное объявление addToken( ) приводит к переполнению на той машине, где тип char является знаковым, всякий раз, когда используется лексема со значением больше 127. Если бы такой программный код компилировался и связывался без ошибки, во время выполнения могли обнаружиться серьезные последствия.

В C++ информация о количестве и типах параметров функций помещается в имя функции – это называется *безопасным связыванием* (type-safe linkage). Оно помогает обнаружить расхождения в объявлениях функций в разных файлах. Поскольку типы параметров unsigned char и char различны, в соответствии с принципом безопасного связывания функция addToken( ), объявленная в файле lex.C, будет считаться неизвестной. Согласно стандарту определение в файле token.C задает другую функцию.

Подобный механизм обеспечивает некоторую степень проверки типов при вызове функций из разных файлов. Безопасное связывание также необходимо для поддержки перегруженных функций. (Мы продолжим рассмотрение этой проблемы в главе 9.)

Прочие типы несоответствия объявлений одного и того же объекта или функции в разных файлах не обнаруживаются во время компиляции или связывания. Поскольку компилятор обрабатывает отдельно каждый файл, он не способен сравнить типы в разных файлах. Несоответствия могут быть источником серьезных ошибок, проявляющихся, подобно приведенным ниже, только во время выполнения программы (к

```
// в token. C
unsigned char lastTok = 0;
unsigned char peekTok() { /* ... */ }

// в lex.C
extern char lastTok;
```

примеру, путем возбуждения исключения или из-за вывода неправильной информации).

```
extern char peekTok();
```

Избежать подобных неточностей поможет прежде всего правильное использование заголовочных файлов. Мы поговорим об этом в следующем подразделе.

8.2.3. Несколько слов о заголовочных файлах

Заголовочный файл предоставляет место для всех extern-объявлений объектов, объявлений функций и определений встроенных функций. Это называется локализацией

объявлений. Те исходные файлы, где объект или функция определяется или используется, должны *включать* заголовочный файл.

Такие файлы позволяют добиться двух целей. Во-первых, гарантируется, что все исходные файлы содержат одно и то же объявление для глобального объекта или функции. Во-вторых, при необходимости изменить объявление это изменение делается в одном месте, что исключает возможность забыть внести правку в какой-то из исходных файлов.

```
// ----- token.h -----
typedef unsigned char uchar;
const uchar INLINE = 128;
// ...
const uchar IT = ...;
const uchar GT = ...;

extern uchar lastTok;
extern int addToken( uchar );
inline bool is_relational( uchar tok )
  { return (tok >= LT && tok <= GT); }

// ----- lex.C -----
#include "token.h"
// ...

// ----- token.C -----
#include  "token.h"
```

Пример с `addToken()` имеет следующий заголовочный файл:

```
// ...
```

При проектировании заголовочных файлов нужно учитывать несколько моментов. Все объявления такого файла должны быть логически связанными. Если он слишком велик или содержит слишком много не связанных друг с другом элементов, программисты не станут включать его, экономя на времени компиляции. Для уменьшения временных затрат в некоторых реализациях C++ предусматривается использование *предкомпилированных* заголовочных файлов. В руководстве к компилятору сказано, как создать такой файл из обычного. Если в вашей программе используются большие заголовочные файлы, применение предкомпиляции может значительно сократить время обработки.

Чтобы это стало возможным, заголовочный файл не должен содержать объявлений встроенных (inline) функций и объектов. Любая из следующих инструкций является

```
extern int ival = 10;
double fica_rate;
```

определением и, следовательно, не может быть использована в заголовочном файле:

```
extern void dummy () {}
```

Хотя переменная `i` объявлена с ключевым словом `extern`, явная инициализация превращает ее объявление в определение. Точно так же и функция `dummy()`, несмотря на явное объявление как `extern`, определяется здесь же: пустые фигурные скобки содержат ее тело. Переменная `fica_rate` определяется и без явной инициализации: об этом

говорит отсутствие ключевого слова extern. Включение такого заголовочного файла в два или более исходных файла одной программы вызовет ошибку связывания – повторные определения объектов.

В файле token.h, приведенном выше, константа INLINE и встроенная функция is_relational() кажутся нарушающими правило. Однако это не так.

Определения символических констант и встроенных функций являются специальными видами определений: те и другие могут появиться в программе несколько раз.

При возможности компилятор заменяет имя символической константы ее значением. Этот процесс называют *подстановкой константы*. Например, компилятор подставит 128 вместо INLINE везде, где это имя встретится в исходном файле. Для того чтобы компилятор произвел такую замену, определение константы (значение, которым она инициализирована) должно быть видимо в том месте, где она используется. Определение символической константы может появиться несколько раз в разных файлах, потому что в результирующем исполняемом файле благодаря подстановке оно будет только одно.

В некоторых случаях, однако, такая подстановка невозможна. Тогда лучше вынести инициализацию константы в отдельный исходный файл. Это делается с помощью явного

```
// ----- заголовочный файл -----
const int buf_chunk = 1024;
extern char *const bufp;

// ----- исходный файл -----
```

объявления константы как extern. Например:

```
char *const bufp = new char[buf_chunk];
```

Хотя bufp объявлена как const, ее значение не может быть вычислено во время компиляции (она инициализируется с помощью оператора new, который требует вызова библиотечной функции). Такая конструкция в заголовочном файле означала бы, что константа определяется каждый раз, когда этот заголовочный файл включается. Символическая константа – это любой объект, объявленный со спецификатором const. Можете ли вы сказать, почему следующее объявление, помещенное в заголовочный файл,

```
// ошибка: не должно быть в заголовочном файле
```

вызывает ошибку связывания, если такой файл включается в два различных исходных?

```
const char* msg = "?? oops: error: ";
```

Проблема вызвана тем, что msg не константа. Это неконстантный указатель, адресующий константу. Правильное объявление выглядит так (полное описание объявлений указателей см. в главе 3):

```
const char *const  msg = "?? oops: error: ";
```

Такое определение может появиться в разных файлах.

Схожая ситуация наблюдается и со встроенными функциями. Для того чтобы компилятор мог подставить тело функции “по месту”, он должен видеть ее определение. (Встроенные функции были представлены в разделе 7.6.)

Следовательно, встроенная функция, необходимая в нескольких исходных файлах, должна быть определена в заголовочном файле. Однако спецификация `inline` – только "совет" компилятору. Будет ли функция встроенной везде или только в данном конкретном месте, зависит от множества обстоятельств. Если компилятор пренебрегает спецификацией `inline`, он генерирует определение функции в исполняемом файле. Если такое определение появится в данном файле больше одного раза, это будет означать ненужную трату памяти.

Большинство компиляторов выдают предупреждение в любом из следующих случаев (обычно это требует включения режима выдачи предупреждений):

- само определение функции не позволяет встроить ее. Например, она слишком сложна. В таком случае попробуйте переписать функцию или уберите спецификацию `inline` и поместите определение функции в исходный файл;
- конкретный вызов функции может не быть "подставлен по месту". Например, в оригинальной реализации C++ компании AT&T (cfront) такая подстановка невозможна для второго вызова в пределах одного и того же выражения. В такой ситуации выражение следует переписать, разделив вызовы встроенных функций.

Перед тем как употребить спецификацию `inline`, изучите поведение функции во время выполнения. Убедитесь, что ее действительно можно встроить. Мы не рекомендуем объявлять функции встроенными и помещать их определения в заголовочный файл, если они не могут быть таковыми по своей природе.

Упражнение 8.3

Установите, какие из приведенных ниже инструкций являются объявлениями, а какие – определениями, и почему:

```
(a) extern int ix = 1024;
(b) int iy;
(c) extern void reset( void *p ) { /* ... */ }
(d) extern const int *pi;
(e) void print( const matrix & );
```

Упражнение 8.4

Какие из приведенных ниже объявлений и определений вы поместили бы в заголовочный файл? В исходный файл? Почему?

```
(a) int var;
(b) inline bool is_equal( const SmallInt &, const SmallInt & ){ }
(c) void putValues( int *arr, int size );
(d) const double pi = 3.1416;
(e) extern int total = 255;
```

8.3. Локальные объекты

Объявление переменной в локальной области видимости вводит *локальный объект*. Существует три вида таких объектов: *автоматические*, *регистровые* и *статические*, различающиеся временем жизни и характеристиками занимаемой памяти.

Автоматический объект существует с момента активизации функции, в которой он определен, до выхода из нее. Регистровый объект – это автоматический объект, для которого поддерживается быстрое считывание и запись его значения. Локальный статический объект располагается в области памяти, существующей на протяжении всего времени выполнения программы. В этом разделе мы рассмотрим свойства всех этих объектов.

8.3.1. Автоматические объекты

Автоматический объект размещается в памяти во время вызова функции, в которой он определен. Память для него отводится из программного стека в записи активации функции. Говорят, что такие объекты имеют *автоматическую продолжительность хранения*, или *автоматическую протяженность*. Неинициализированный автоматический объект содержит случайное, или *неопределенное*, значение, оставшееся от предыдущего использования области памяти. После завершения функции ее запись активации выталкивается из программного стека, т.е. память, ассоциированная с локальным объектом, освобождается. Время жизни такого объекта заканчивается с завершением работы функции, и его значение теряется.

Поскольку память, отведенная локальному объекту, освобождается при завершении работы функции, адрес автоматического объекта следует использовать с осторожностью. Например, этот адрес не может быть возвращаемым значением, так как после

```
#include "Matrix.h"

Matrix* trouble( Matrix *pm )
{
    Matrix res;
    // какие-то действия
    // результат присвоим res

    return &res; // плохо!
}

int main()
{
    Matrix m1;
    // ...
    Matrix *mainResult = trouble( &m1 );
    // ...
```

выполнения функции будет относиться к несуществующему объекту:

```
}
```

`mainResult` получает значение адреса автоматического объекта `res`. К несчастью, память, отведенная под `res`, освобождается по завершении функции `trouble()`. После возврата в `main()` `mainResult` указывает на область памяти, не отведенную никакому объекту. (В данном примере эта область все еще может содержать правильное значение, поскольку мы не вызывали других функций после `trouble()` и запись ее активации, вероятно, еще не затерта.) Подобные ошибки обнаружить весьма трудно. Дальнейшее использование `mainResult` в программе скорее всего даст неверные результаты.

Передача в функцию `trouble()` адреса `m1` автоматического объекта функции `main()` безопасна. Память, отведенная `main()`, во время вызова `trouble()`находится в стеке, так что `m1` остается доступной внутри `trouble()`.

Если адрес автоматического объекта сохраняется в указателе, время жизни которого больше, чем самого объекта, такой указатель называют *висячим*. Работа с ним – это серьезная ошибка, поскольку содержимое адресуемой области памяти непредсказуемо. Если комбинация бит по этому адресу оказывается в какой-то степени допустимой (не приводит к нарушению защиты памяти), то программа будет выполняться, но результаты ее будут неправильными.

8.3.2. Регистровые автоматические объекты

Автоматические объекты, интенсивно используемые в функции, можно объявить с ключевым словом `register`, тогда компилятор будет их загружать в машинные регистры. Если же это невозможно, объекты останутся в основной памяти. Индексы массивов и указатели, встречающиеся в циклах, – хорошие кандидаты в регистровые

```
for ( register int ix =0; ix < sz; ++-ix ) // ...
```

объекты.

```
for ( register int *p = array ; p < arraySize; ++p ) // ...
```

```
bool find( register int *pm, int Val ) {
    while ( *pm )
        if ( *pm++ == Val ) return true;
    return false;
}
```

Параметры также можно объявлять как регистровые переменные:

Их активное использование может заметно увеличить скорость выполнения функции.

Указание ключевого слова `register` – только подсказка компилятору. Некоторые компиляторы игнорируют такой запрос, применяя специальные алгоритмы для определения наиболее подходящих кандидатов на размещение в свободных регистрах.

Поскольку компилятор учитывает архитектуру машины, на которой будет выполняться программа, он зачастую может принять более обоснованное решение об использовании машинных регистров.

8.3.3. Статические локальные объекты

Внутри функции или составной инструкции можно объявить объект с локальной областью видимости, который, однако, будет существовать в течение всего времени выполнения программы. Если значение локального объекта должно сохраняться между вызовами функции, то обычный автоматический объект не подойдет: ведь его значение теряется каждый раз после выхода.

В таком случае локальный объект необходимо объявить как `static` (со *статической продолжительностью хранения*). Хотя значение такого объекта сохраняется между вызовами функции, в которой он определен, видимость его имени ограничена локальной областью. Статический локальный объект инициализируется во время первого

выполнения инструкции, где он объявлен. Вот, например, версия функции

```
#include <iostream>

int traceGcd( int v1, int v2 )
{
    static int depth = 1;
    cout << "глубина #" << depth++ << endl;

    if ( v2 == 0 ) {
        depth = 1;
        return v1;
    }
    return traceGcd( v2, v1%v2 );
```

gcd(),устанавливающая глубину рекурсии с его помощью:

```
}
```

Значение, ассоциированное со статическим локальным объектом depth, сохраняется между вызовами traceGcd(). Его инициализация выполняется только один раз – когда к

```
#include <iostream>
extern int traceGcd(int, int);

int main() {
    int rslt = traceCcd( 15, 123 );
    cout << "НОД (15,123): " << rslt << endl;
    return 0;
```

этой функции обращаются впервые. В следующей программе используется traceGcd():

```
}
```

Результат работы программы:

```
глубина #1
глубина #2
глубина #3
глубина #4
НОД (15,123): 3
```

Неинициализированные статические локальные объекты получают значение 0. А автоматические объекты в подобной ситуации получают случайные значения. Следующая программа иллюстрирует разницу инициализации по умолчанию для автоматических и статических объектов и опасность, подстерегающую программиста в случае ее отсутствия для автоматических объектов.

```
#include <iostream>

const int iterations = 2;
void func() {
    int value1, value2; // не инициализированы
    static int depth;   // неявно инициализирован нулем

    if ( depth < iterations )
        { ++depth; func(); }
    else depth = 0;

    cout << "\nvaluel:\t" << value1;
    cout << "\tvalue2:\t" << value2;
    cout << "\tsum:\t" << value1 + value2;
}

int main() {
    for ( int ix = 0; ix < iterations; ++ix ) func();
    return 0;
}
```

Вот результат работы программы:

```
valuel:  0          value2:  74924      sum:  74924
valuel:  0          value2:  68748      sum:  68748
valuel:  0          value2:  68756      sum:  68756
valuel:  148620     value2:  2350       sum:  150970
valuel:  2147479844 value2:  671088640  sum:  -1476398812
valuel:  0          value2:  68756      sum:  68756
```

value1 и value2 – неинициализированные автоматические объекты. Их начальные значения, как можно видеть из приведенной распечатки, оказываются случайными, и потому результаты сложения непредсказуемы. Объект depth, несмотря на отсутствие явной инициализации, гарантированно получает значение 0, и функция func() рекурсивно вызывает сама себя только дважды.

8.4. Динамически размещаемые объекты

Время жизни глобальных и локальных объектов четко определено. Программист неспособен хоть как-то изменить его. Однако иногда необходимо иметь объекты, временем жизни которых можно управлять. Выделение памяти под них и ее освобождение зависят от действий выполняющейся программы. Например, можно отвести память под текст сообщения об ошибке только в том случае, если ошибка действительно имела место. Если программа выдает несколько таких сообщений, размер выделяемой строки будет разным в зависимости от длины текста, т.е. подчиняется типу ошибки, произошедшей во время исполнения программы.

Третий вид объектов позволяет программисту полностью управлять выделением и освобождением памяти. Такие объекты называют *динамически размещаемыми* или, для краткости, просто *динамическими*. Динамический объект “живет” в пуле свободной памяти, называемой *хипом*. Программист создает его с помощью оператора new, а уничтожает с помощью оператора delete. Динамически размещаться может как единичный объект, так и массив объектов. Размер массива, размещаемого в хипе, разрешается задавать во время выполнения.

В этом разделе, посвященном динамическим объектам, мы рассмотрим три формы оператора new: для размещения единичного объекта, для размещения массива и третью форму, называемую *оператором размещения* new (placement new expression). Когда хип исчерпан, этот оператор возбуждает исключение. (Разговор об исключениях будет продолжен в главе 11. В главе 15 мы расскажем об операторах new и delete применительно к классам.)

8.4.1. Динамическое создание и уничтожение единичных объектов

Оператор new состоит их ключевого слова new, за которым следует спецификатор типа. Этот спецификатор может относиться к встроенным типам или к типам классов. Например:

```
new int;
```

размещает в хипе один объект типа int. Аналогично в результате выполнения инструкции

```
new iStack;
```

там появится один объект класса iStack.

Сам по себе оператор new не слишком полезен. Как можно реально воспользоваться созданным объектом? Одним из аспектов работы с памятью из хипа является то, что размещаемые в ней объекты не имеют имени. Оператор new возвращает не сам объект, а указатель на него. Все манипуляции с этим объектом производятся косвенно через указатели:

```
int *pi = new int;
```

Здесь оператор new создает один объект типа int, на который ссылается указатель pi. Выделение памяти из хипа во время выполнения программы называется *динамическим выделением*. Мы говорим, что память, адресуемая указателем pi, выделена динамически.

Второй аспект, относящийся к использованию хипа, состоит в том, что эта память не инициализируется. Она содержит “мусор”, оставшийся после предыдущей работы. Проверка условия:

```
if ( *pi == 0 )
```

вероятно, даст false, поскольку объект, на который указывает pi, содержит случайную последовательность битов. Следовательно, объекты, создаваемые с помощью оператора new, рекомендуется инициализировать. Программист может инициализировать объект типа int из предыдущего примера следующим образом:

```
int *pi = new int( 0 );
```

Константа в скобках задает начальное значение для создаваемого объекта; теперь pi ссылается на объект типа int, имеющий значение 0. Выражение в скобках называется *инициализатором*. Это может быть любое выражение (не обязательно константа), возвращающее значение, приводимое к типу int.

Оператор new выполняет следующую последовательность действий: выделяет из хипа память для объекта, затем инициализирует его значением, стоящим в скобках. Для выделения памяти вызывается библиотечная функция new(). Предыдущий оператор

```
int ival = 0;    // создаем объект типа int и инициализируем его 0
```

приблизительно эквивалентен следующей последовательности инструкций:

```
int *pi = &ival; // указатель ссылается на этот объект
```

не считая, конечно, того, что объект, адресуемый pi, создается библиотечной функцией new() и размещается в хипе. Аналогично

```
iStack *ps = new iStack( 512 );
```

создает объект типа iStack на 512 элементов. В случае объекта класса значение или значения в скобках передаются соответствующему конструктору, который вызывается в случае успешного выделения памяти. (Динамическое создание объектов классов более подробно рассматривается в разделе 15.8. Оставшаяся часть данного раздела посвящена созданию объектов встроенных типов.)

Описанные операторы new могут вызывать одну проблему: хип, к сожалению, является конечным ресурсом, и в некоторой точке выполнения программы мы можем исчерпать его. Если функция new() не может выделить затребованного количества памяти, она возбуждает исключение bad_alloc. (Обработка исключений рассматривается в главе 11.)

Время жизни объекта, на который указывает pi, заканчивается при освобождении памяти, где этот объект размещен. Это происходит, когда pi передается оператору delete. Например,

```
delete pi;
```

освобождает память, на которую ссылается pi, завершая время жизни объекта типа int. Программист управляет окончанием жизни объекта, используя оператор delete в нужном месте программы. Этот оператор вызывает библиотечную функцию delete(), которая возвращает выделенную память в хип. Поскольку хип конечен, очень важно возвращать ее своевременно.

Глядя на предыдущий пример, вы можете спросить: а что случится, если значение pi по

```
// необходимо ли это?
if ( pi != 0 )
```

какой-либо причине было нулевым? Не следует ли переписать этот код таким образом:

```
    delete pi;
```

Нет. Язык C++ гарантирует, что оператор delete не будет вызывать функцию delete() в случае нулевого операнда. Следовательно, проверка на 0 необязательна. (Если вы явно добавите такую проверку, в большинстве реализаций она фактически будет выполнена дважды.)

Важно понимать разницу между временем жизни указателя pi и объекта, который он адресует. Сам объект pi является глобальным и объявлен в глобальной области

видимости. Следовательно, память под него выделяется до выполнения программы и сохраняется за ним до ее завершения. Совсем не так определяется время жизни адресуемого указателем pi объекта, который создается с помощью оператора new во время выполнения. Область памяти, на которую указывает pi, выделена динамически, следовательно, pi является указателем на динамически размещенный объект типа int. Когда в программе встретится оператор delete, эта память будет освобождена. Однако память, отведенная самому указателю pi, не освобождается, а ее содержимое не изменяется. После выполнения delete объект pi становится висячим указателем, то есть ссылается на область памяти, не принадлежащую программе. Такой указатель служит источником трудно обнаруживаемых ошибок, поэтому сразу после уничтожения объекта ему полезно присвоить 0, обозначив таким образом, что указатель больше ни на что не ссылается.

Оператор delete может использоваться только по отношению к указателю, который содержит адрес области памяти, выделенной в результате выполнения оператора new. Попытка применить delete к указателю, не ссылающемуся на такую память, приведет к непредсказуемому поведению программы. Однако, как было сказано выше, этот оператор можно применять к нулевому указателю.

```
void f() {
    int i;
    string str = "dwarves";
    int *pi = &i;
    short *ps = 0;
    double *pd = new doub1e(33);

    delete str; // плохо: str не является динамическим объектом
    delete pi; // плохо: pi ссылается на локальный объект
    delete ps; // безопасно
    delete pd; // безопасно
}
```

Ниже приведены примеры опасных и безопасных операторов delete:

Вот три основные ошибки, связанные с динамическим выделением памяти:

- не освободить выделенную память. В таком случае память не возвращается в хип. Эта ошибка получила название *утечки памяти*;
- дважды применить оператор delete к одной и той же области памяти. Такое бывает, когда два указателя получают адрес одного и того же динамически размещенного объекта. В результате подобной ошибки мы вполне можем удалить нужный объект. Действительно, память, освобожденная с помощью одного из адресующих ее указателей, возвращается в хип и затем выделяется под другой объект. Затем оператор delete применяется ко второму указателю, адресовавшему старый объект, а удаляется при этом новый;
- изменять объект после его удаления. Такое часто случается, поскольку указатель, к которому применяется оператор delete, не обнуляется.

Эти ошибки при работе с динамически выделяемой памятью гораздо легче допустить, нежели обнаружить и исправить. Для того чтобы помочь программисту, стандартная библиотека C++ представляет класс auto_ptr. Мы рассмотрим его в следующем подразделе. После этого мы покажем, как динамически размещать и уничтожать массивы, используя вторую форму операторов new и delete.

8.4.2. Шаблон auto_ptr A

В стандартной библиотеке C++ auto_ptr является шаблоном класса, призванным помочь программистам в манипулировании объектами, которые создаются посредством оператора new. (К сожалению, подобного шаблона для манипулирования динамическими массивами нет. Использовать auto_ptr для создания массивов нельзя, это приведет к непредсказуемым результатам.)

Объект auto_ptr инициализируется адресом динамического объекта, созданного с помощью оператора new. Такой объект автоматически уничтожается, когда заканчивается время жизни auto_ptr. В этом подразделе мы расскажем, как ассоциировать auto_ptr с динамически размещаемыми объектами.

Для использования шаблона класса auto_ptr необходимо включить заголовочный файл:

```
#include <memory>
```

```
auto_ptr< type_pointed_to > identifier( ptr_allocated_by_new );
auto_ptr< type_pointed_to > identifier( auto_ptr_of_same_type );
```

Определение объекта auto_ptr имеет три формы:

```
auto_ptr< type_pointed_to > identifier;
```

Здесь type_pointed_to представляет собой тип нужного объекта. Рассмотрим последовательно каждое из этих определений. Как правило, мы хотим непосредственно инициализировать объект auto_ptr адресом объекта, созданного с помощью оператора new. Это можно сделать следующим образом:

```
auto_ptr< int > pi ( new int( 1024 ) );
```

В результате значением pi является адрес созданного объекта, инициализированного числом 1024. С объектом, на который указывает auto_ptr, можно работать обычным

```
if ( *pi != 1024 )
    // ошибка, что-то не так
```

способом:

```
else *pi *= 2;
```

Объект, на который указывает pi, будет автоматически уничтожен по окончании времени жизни pi. Если указатель pi является локальным, то объект, который он адресует, будет уничтожен при выходе из блока, где он определен. Если же pi глобальный, то объект, на который он ссылается, уничтожается при выходе из программы.

Что будет, если мы инициализируем auto_ptr адресом объекта класса, скажем,

```
auto_ptr< string >
```

стандартного класса string? Например:

```
    pstr_auto( new string( "Brontosaurus" ) );
```

Предположим, что мы хотим выполнить какую-то операцию со строками. С обычной

```
string *pstr_type = new string( "Brontosaurus" );
if ( pstr_type->empty() )
```

строкой мы бы поступили таким образом:

```
    // ошибка, что-то не так
```

```
auto_ptr< string > pstr_auto( new    string( "Brontosaurus" ) );
if ( pstr_type->empty() )
```

А как обратиться к операции empty(), используя объект auto_ptr? Точно так же:

```
    // ошибка, что-то не так
```

Создатели шаблона класса auto_ptr не в последнюю очередь стремились сохранить привычный синтаксис, употребляемый с обычными указателями, а также обеспечить дополнительные возможности автоматического удаления объекта, на который ссылается auto_ptr. При этом время выполнения не увеличивается. Применение встроенных функций (которые подставляются по месту вызова) позволило сделать использование объекта auto_ptr немногим более дорогим, чем непосредственное употребление указателя.

Что произойдет, если мы проинициализируем pstr_auto2 значением pstr_auto,

```
// кто несет ответственность за уничтожение строки?
```

который является объектом auto_ptr, указывающим на строку?

```
auto_ptr< string > pstr_auto2( pstr_auto );
```

Представим, что мы непосредственно инициализировали один указатель на строку другим:

```
string *pstr_type2( pstr_type );
```

Оба указателя теперь содержат адрес одной и той же строки, и мы должны быть внимательными, чтобы не удалить строку дважды.

В противоположность этому шаблон класса auto_ptr поддерживает понятие *владения*. Когда мы определили pstr_auto, он стал владельцем строки, адресом которой был инициализирован, и принял на себя ответственность за ее уничтожение.

Вопрос в том, кто станет владельцем строки, когда мы инициализируем pstr_auto2 адресом, указывающим на тот же объект, что и pstr_auto? Нежелательно, чтобы оба объекта владели одной и той же строкой: это вернет нас к проблемам повторного удаления, от которых мы стремились уйти с помощью шаблона класса auto_ptr.

Когда один объект auto_ptr инициализируется другим или получает его значение в результате присваивания, одновременно он получает и право владения адресуемым объектом. Объект auto_ptr, стоящий справа от оператора присваивания, передает право

владения и ответственность auto_ptr, стоящему слева. В нашем примере ответственность за уничтожение строки несет pstr_auto2, а не pstr_auto. pstr_auto больше не может употребляться для ссылки на эту строку.

```
auto_ptr< int > p1( new int( 1024 ) );
```

Аналогично ведет себя и операция присваивания. Пусть у нас есть два объекта auto_ptr:

```
auto_ptr< int > p2( new int( 2048 ) );
```

Мы можем скопировать один объекта auto_ptr в другой с помощью этой операции:

```
p1 = p2;
```

Перед присваиванием объект, на который ссылался p1, удаляется.

После присваивания p1 владеет объектом типа int со значением 2048. p2 больше не может использоваться как ссылка на этот объект.

Третья форма определения объекта auto_ptr создает его, но не инициализирует

```
// пока не ссылается ни на какой объект
```

значением указателя на область памяти из хипа. Например:

```
auto_ptr< int > p_auto_int;
```

Поскольку p_auto_int не инициализирован адресом какого-либо объекта, значение хранящегося внутри него указателя равно 0. Разыменование таких указателей приводит к

```
// ошибка: разыменование нулевого указателя
if ( *p_auto_int != 1024 )
```

непредсказуемому поведению программы:

```
    *p_auto_int = 1024;
```

```
int *pi = 0;
```

Обычный указатель можно проверить на равенство 0:

```
if ( pi ! = 0 ) ...;
```

А как проверить, адресует auto_ptr какой-либо объект или нет? Операция get() возвращает внутренний указатель, использующийся в объекте auto_ptr. Значит, мы

```
// проверяем, указывает ли p_auto_int на объект
if ( p_auto_int.get() != 0 &&
    *p_auto_int != 1024 )
```

должны применить следующую проверку:

```
    *p_auto_int = 1024;
```

Если auto_ptr ни на что не указывает, то как заставить его адресовать что-либо? Другими словами, как мы можем присвоить значение внутреннему указателю объекта

```
else
   // хорошо, присвоим ему значение
```

auto_ptr? Это делается с помощью операции reset(). Например:

```
   p_auto_int.reset( new int( 1024 ) );
```

Объекту auto_ptr нельзя присвоить адрес объекта, созданного с помощью оператора

```
void example() {
    // инициализируется нулем по умолчанию
    auto_ptr< int > pi;
    {
        // не поддерживается
        pi = new int( 5 ) ;
    }
```

new:

```
}
```

В этом случае надо использовать функцию reset(), которой можно передать указатель или 0, если мы хотим обнулить объект auto_ptr. Если auto_ptr указывает на объект и является его владельцем, то этот объект уничтожается перед присваиванием нового

```
auto_ptr< string >
    pstr_auto( new string( "Brontosaurus" ) );

// "Brontosaurus" уничтожается перед присваиванием
```

значения внутреннему указателю auto_ptr. Например:

```
pstr_auto.reset( new string( "Long-neck" ) );
```

В последнем случае лучше, используя операцию assign(), присвоить новое значение

```
// более эффективный способ присвоить новое значение

// используем операцию assign()
```

существующей строке, чем уничтожать одну строку и создавать другую:

```
pstr_auto->assign( "Long-neck" );
```

Одна из трудностей программирования состоит в том, что получить правильный результат не всегда достаточно. Иногда накладываются и временные ограничения. Такая мелочь, как удаление и создание заново строкового объекта, вместо использования функции assign() при определенных обстоятельствах может вызвать значительное замедление работы. Подобные детали не должны вас беспокоить при проектировании, но при доводке программы на них следует обращать внимание.

Шаблон класса auto_ptr обеспечивает значительные удобства и безопасность использования динамически выделяемой памяти. Однако все равно надо не терять бдительности, чтобы не навлечь на себя неприятности:

- нельзя инициализировать объект auto_ptr указателем, полученным не с помощью оператора new, или присвоить ему такое значение. В противном случае после применения к этому объекту оператора delete поведение программы непредсказуемо;
- два объекта auto_ptr не должны получать во владение один и тот же объект. Очевидный способ допустить такую ошибку – присвоить одно значение двум

```
auto_ptr< string >
    pstr_auto( new string( "Brontosaurus" ) );
// ошибка: теперь оба указывают на один объект
// и оба являются его владельцами
```

объектам. Менее очевидный – с помощью операции get(). Вот пример:

```
auto_ptr< string > pstr_auto2( pstr_auto.get() );
```

Операция release() гарантирует, что несколько указателей не являются владельцами одного и того же объекта. release() не только возвращает адрес объекта, на который ссылается auto_ptr, но и передает владение им.

```
// правильно: оба указывают на один объект,
// но pstr_auto больше не является его владельцем
auto_ptr< string >
```

Предыдущий фрагмент кода нужно переписать так:

```
pstr_auto2( pstr_auto.release() );
```

8.4.3. Динамическое создание и уничтожение массивов

Оператор new может выделить из хипа память для размещения массива. В этом случае после спецификатора типа в квадратных скобках указывается размер массива. Он может быть задан сколь угодно сложным выражением. new возвращает указатель на первый

```
// создание единственного объекта типа int
// с начальным значением 1024
int *pi = new int( 1024 );

// создание массива из 1024 элементов
// элементы не инициализируются
int *pia = new int[ 1024 ];

// создание двумерного массива из 4x1024 элементов
```

элемент массива. Например:

```
int (*pia2)[ 1024 ] = new int[ 4 ][ 1024 ];
```

pi содержит адрес единственного элемента типа int, инициализированного значением 1024; pia – адрес первого элемента массива из 1024 элементов; pia2 – адрес начала

массива, содержащего четыре массива по 1024 элемента, т.е. pia2 адресует 4096 элементов.

В общем случае массив, размещаемый в хипе, не может быть инициализирован. (В разделе 15.8 мы покажем, как с помощью конструктора по умолчанию присвоить начальное значение динамическому массиву объектов типа класса.) Задавать инициализатор при выделении оператором new памяти под массив не разрешается. Массиву элементов встроенного типа, размещенному в хипе, начальные значения

```
for (int index = 0; index < 1024; ++index )
```

присваиваются с помощью цикла for:

```
    pia[ index ] = 0;
```

Основное преимущество динамического массива состоит в том, что количество элементов в его первом измерении не обязано быть константой, т.е. может не быть известным во время компиляции. Для массивов, определяемых в локальной или глобальной области видимости, это не так: здесь размер задавать необходимо.

Например, если указатель в ходе выполнения программы ссылается на разные С-строки, то область памяти под текущую строку обычно выделяется динамически и ее размер определяется в зависимости от длины строки. Как правило, это более эффективно, чем создавать массив фиксированного размера, способный вместить самую длинную строку: ведь все остальные строки могут быть значительно короче. Более того, программа может аварийно завершиться, если длина хотя бы одной из строк превысит отведенный лимит.

Оператор new допустимо использовать для задания первого измерения массива с помощью значения, вычисляемого во время выполнения. Предположим, у нас есть

```
const char *noerr = "success";
// ...
const char *err189 = "Error: a function declaration must "
```

следующие С-строки:

```
                      "specify a function return type!";
```

Размер создаваемого с помощью оператора new массива может быть задан значением,

```
#include <cstring>

const char *errorTxt;

if (errorFound)
    errorTxt = errl89;
else
    errorTxt = noerr;

int dimension = strlen( errorTxt ) + 1;
char *strl = new char[ dimension ];

// копируем текст ошибки в strl
```

вычисляемым во время выполнения:

```
strcpy( strl, errorTxt );
```

```
// обычная для C++ идиома,
// иногда удивляющая начинающих программистов
```

dimension разрешается заменить выражением:

```
char *strl = new char[ str1en( errorTxt ) + 1 ];
```

Единица, прибавляемая к значению, которое возвращает strlen(), необходима для учета завершающего нулевого символа в С-строке. Отсутствие этой единицы – весьма распространенная ошибка, которую достаточно трудно обнаружить, поскольку она проявляет себя косвенно: происходит затирание какой-либо другой области программы. Почему? Большинство функций, которые обрабатывают массивы, представляющие собой С-строки символов, пробегают по элементам, пока не встретят завершающий нуль.

Если в конце строки нуля нет, то возможно чтение или запись в случайную область памяти. Избежать подобных проблем позволяет класс string из стандартной библиотеки C++.

Отметим, что только первое измерение массива, создаваемого с помощью оператора new, может быть задано значением, вычисляемым во время выполнения. Остальные измерения

```
int getDim();

// создание двумерного массива
int (*pia3)[ 1024 ] = new int[ getDim() ][ 1024 ]; // правильно

// ошибка: второе измерение задано не константой
```

должны задаваться константами, известными во время компиляции. Например:

```
int **pia4 = new int[ 4 ][ getDim() ];
```

Оператор delete для уничтожения массива имеет следующую форму:

```
delete[] str1;
```

Пустые квадратные скобки необходимы. Они говорят компилятору, что указатель адресует массив, а не единичный элемент. Поскольку тип str1 – указатель на char, без этих скобок компилятор не поймет, что удалять следует целый массив.

Отсутствие скобок не является синтаксической ошибкой, но правильность выполнения программы не гарантируется (это особенно справедливо для массивов, которые содержат объекты классов, имеющих деструкторы, как это будет показано в разделе 14.4).

Чтобы избежать проблем, связанных с управлением динамически выделяемой памятью для массивов, рекомендуется пользоваться контейнерными типами из стандартной библиотеки, такими, как vector, list или string. Они управляют памятью автоматически. (Тип string был представлен в разделе 3.4, тип vector – в разделе 3.10. Подробное описание контейнерных типов см. в главе 6.)

8.4.4. Динамическое создание и уничтожение константных объектов

Программист способен создать объект в хипе и запретить изменение его значения после инициализации. Этого можно достичь, объявляя объект константным. Для этого применяется следующая форма оператора new:

```
const int *pci = new const int(1024);
```

Константный динамический объект имеет несколько особенностей. Во-первых, он должен быть инициализирован, иначе компилятор сигнализирует об ошибке (кроме случая, когда объект принадлежит к типу класса, имеющего конструктор по умолчанию; в такой ситуации инициализатор можно опустить).

Во-вторых, указатель, возвращаемый выражением new, должен адресовать константу. В предыдущем примере pci служит указателем на const int.

Константность динамически созданного объекта подразумевает, что значение, полученное при инициализации, в дальнейшем не может быть изменено. Но поскольку объект динамический, временем его жизни управляет оператор delete. Например:

```
delete pci;
```

Хотя операнд оператора delete имеет тип указателя на const int, эта инструкция является корректной и освобождает область памяти, на которую ссылается pci.

Невозможно создать динамический массив константных элементов встроенного типа потому, что, как мы отмечали выше, элементы такого массива нельзя проинициализировать в операторе new. Следующая инструкция приводит к ошибке компиляции:

```
const int *pci = new const int[100]; // ошибка
```

8.4.5. Оператор размещения new A

Существует третья форма оператора new, которая создает объект без отведения для него памяти, то есть в памяти, которая уже была выделена. Эту форму называют *оператором размещения* new. Программист указывает адрес области памяти, в которой размещается объект:

```
new (place_address) type-specifier
```

place_address должен быть указателем. Такая форма (она включается заголовочным файлом <new>) позволяет программисту предварительно выделить большую область памяти, которая впоследствии будет содержать различные объекты. Например:

```
#include <iostream>
#include <new>
const int chunk = 16;
class Foo {
public:
    int val() { return _val; }
    FooQ(){ _val = 0; }
private:
    int _val;
};
// выделяем память, но не создаем объектов Foo
char *buf = new char[ sizeof(Foo) * chunk ];

int main() {
    // создаем объект Foo в buf
    Foo *pb = new (buf) Foo;

    // проверим, что объект помещен в buf
    if ( pb.val() == 0 )
        cout << "Оператор new сработал!" << endl;

    // здесь нельзя использовать pb
    delete[] buf;

    return 0;
}
```

Результат работы программы:

```
Оператор new сработал!
```

Для оператора размещения new нет парного оператора delete: он не нужен, поскольку эта форма не выделяет память. В предыдущем примере необходимо освободить память, адресуемую указателем buf, а не pb. Это происходит в конце программы, когда буфер больше не нужен. Поскольку buf ссылается на символьный массив, оператор delete имеет форму

```
delete[] buf;
```

При уничтожении buf прекращают существование все объекты, созданные в нем. В нашем примере pb больше не ссылается на существующий объект класса Foo.

Упражнение 8.5

```
(a) const float *pf = new const float[100];
(b) double *pd = new doub1e[10] [getDim()];
(c) int (*pia2)[ 1024 ] = new int[ ][ 1024 ];
```

Объясните, почему приведенные операторы new ошибочны:

```
(d) const int *pci = new const int;
```

Упражнение 8.6

Как бы вы уничтожили pa?

```
typedef int arr[10];
int *pa = new arr;
```

Упражнение 8.7

Какие из следующих операторов `delete` содержат потенциальные ошибки времени

```
int globalObj;
char buf[1000];

void f() {
    int *pi = &global0bj;
    double *pd = 0;
    float *pf = new float(O);
    int *pa = new(buf)int[20];

    delete pi;   // (a)
    delete pd;   // (b)
    delete pf;   // (c)
    de1ete[] pa; // (d)
}
```

выполнения и почему:

Упражнение 8.8

Какие из данных объявлений `auto_ptr` неверны или грозят ошибками времени

```
int ix = 1024;
int *pi = & ix;
int *pi2 = new int ( 2048 );

(a) auto_ptr<int> p0(ix);
(b) auto_ptr<int> pl(pi);
(c) auto_ptr<int> p2(pi2);
(d) auto_ptr<int> p3(&ix);
(e) auto_ptr<int> p4(new int(2048));
(f) auto_ptr<int> p5(p2.get());
(9) auto_ptr<int> p6(p2.release());
```

выполнения? Объясните каждый случай.

```
(h) auto_ptr<int> p7(p2);
```

Упражнение 8.9

```
int *pi0 = p2.get();
```

Объясните разницу между следующими инструкциями:

```
int *pi1 = p2.release() ;
```

Для каких случаев более приемлем тот или иной вызов?

Упражнение 8.10

Пусть мы имеем:

```
auto_ptr< string > ps( new string( "Daniel" )     );
```

В чем разница между этими двумя вызовами assign()?Какой их них предпочтительнее

```
ps.get()->assign( "Danny" );
```

и почему?

```
ps->assign( "Danny" );
```

8.5. Определения пространства имен A

По умолчанию любой объект, функция, тип или шаблон, объявленный в глобальной области видимости, также называемой *областью видимости глобального пространства имен*, вводит *глобальную сущность*. Каждая такая сущность обязана иметь уникальное имя. Например, функция и объект не могут быть одноименными, даже если они объявлены в разных исходных файлах.

Таким образом, используя в своей программе некоторую библиотеку, мы должны быть уверены, что имена глобальных сущностей нашей программы не совпадают с именами из библиотеки. Это нелегко, если мы работаем с библиотеками разных производителей, где определено много глобальных имен. Собирая программу с такими библиотеками, нельзя гарантировать, что имена глобальных сущностей не будут вступать в конфликт.

Обойти эту проблему, названную *проблемой засорения области видимости глобального пространства имен*, можно посредством очень длинных имен. Часто в качестве их

```
class cplusplus_primer_matrix { ... };
```

префикса употребляется определенная последовательность символов. Например:

```
void inverse( cplusplus_primer_matrix & );
```

Однако у этого решения есть недостаток. Программа, написанная на C++, может содержать множество глобальных классов, функций и шаблонов, видимых в любой точке кода. Работать со слишком длинными идентификаторами для программистов утомительно.

Пространства имен помогают справиться с проблемой засорения более удобным способом. Автор библиотеки может задать собственное пространство и таким образом

```
namespace cplusplus_primer {
    class matrix { /*...*/ };
    void inverse ( matrix & );
}
```

вынести используемые в библиотеке имена из глобальной области видимости:

`cplusplus_primer` является *пользовательским пространством имен* (в отличие от глобального пространства, которое неявно подразумевается и существует в любой программе).

Каждое такое пространство представляет собой отдельную область видимости. Оно может содержать вложенные определения пространств имен, а также объявления или определения функций, объектов, шаблонов и типов. Все сущности, объявленные внутри некоторого пространства имен, называются его *членами*. Каждое имя в пользовательском пространстве, как и в глобальном, должно быть уникальным в пределах этого пространства.

Однако в разных пользовательских пространствах могут встречаться члены с одинаковыми именами.

Имя члена пространства имен автоматически дополняется, или *квалифицируется*, именем этого пространства. Например, имя класса `matrix`, объявленное в пространстве `cplusplus_primer`, становится `cplusplus_primer::matrix`, а имя функции `inverse()` превращается в `cplusplus_primer::inverse()`.

Члены `cplusplus_primer` могут использоваться в программе с помощью спецификации

```
void func( cplusplus_primer::matrix &m )
{
    // ...
    cplusplus_primer::inverse(m);
    return m;
}
```

имени:

Если в другом пользовательском пространстве имен (скажем, `DisneyFeatureAnimation`) также существует класс `matrix` и функция `inverse()` и мы хотим использовать этот класс вместо объявленного в пространстве `cplusplus_primer`, то функцию `func()`

```
void func( DisneyFeatureAnimation::matrix &m )
{
    // ...
    DisneyFeatureAnimation::inverse(m);
    return m;
}
```

нужно модифицировать следующим образом:

Конечно, каждый раз указывать специфицированные имена типа

```
namespace_name::member_name
```

неудобно. Поэтому существуют механизмы, позволяющие облегчить использование пространств имен в программах. Это *псевдонимы пространств имен*, *using-объявления* и *using-директивы*. (Мы рассмотрим их в разделе 8.6.)

8.5.1. Определения пространства имен

Определение пользовательского пространства имен начинается с ключевого слова namespace, за которым следует идентификатор. Он должен быть уникальным в той области видимости, в которой определяется данное пространство; наличие другой сущности с тем же именем является ошибкой. Конечно, это не означает, что проблема засорения глобального пространства решена полностью, но существенно помогает в ее решении.

За идентификатором пространства имен следует блок в фигурных скобках, содержащий различные объявления. Любое объявление, допустимое в области видимости глобального пространства, может встречаться и в пользовательском: классы, переменные (вместе с инициализацией), функции (вместе со своими определениями), шаблоны.

Помещая объявление в пользовательское пространство, мы не меняем его семантики. Единственное отличие состоит в том, что имена, вводимые такими объявлениями,

```
namespace cplusplus_primer {
    class matrix { /* ... */ };
    void inverse ( matrix & );
    matrix operator+ ( const matrix &ml, const matrix &m2 )
        {/* ... */ }
    const double pi = 3.1416;
}
```

включают в себя имя пространства, внутри которого они объявлены. Например:

Именем класса, объявленного в пространстве cplusplus_primer, будет

```
cplusplus_primer::matrix
```

Именем функции

```
cplusplus_primer::inverse()
```

Именем константы

```
cplusplus_primer::pi
```

Имя класса, функции или константы расширяется именем пространства, в котором они объявлены. Такие имена называют *квалифицированными*.

Определение пространства имен не обязательно должно быть непрерывным. Например, предыдущее пространство могло быть определено таким образом:

```
namespace cplusplus_primer {
    class matrix { /* ... */ };
    const double pi = 3.1416;
}

namespace cplusplus_primer {
    void inverse ( matrix & );
    matrix operator+ ( const matrix &ml, const matrix &m2 )
        {/* ... */ }

}
```

Два приведенных примера эквивалентны: оба задают пространство имен `cplusplus_primer`, содержащее класс `matrix`, функцию `inverse()`, константу `pi` и `operator+()`. Определение пространства имен может состоять из нескольких соединенных частей.

Последовательность

```
namespace namespace_name {
```

задает новое пространство, если имя `namespace_name` не совпадает с одним из ранее объявленных. В противном случае новые объявления добавляются в старое пространство.

Возможность разбить пространство имен на несколько частей помогает при организации библиотеки. Ее исходный код легко разделить на интерфейсную часть и реализацию.

```
// Эта часть пространства имен
// определяет интерфейс библиотеки

namespace cplusplus_primer {
    class matrix { /* ... */ };
    const double pi = 3.1416;
    matrix operator+ ( const matrix &ml,   const matrix &m2    );
    void inverse ( matrix & );
}

// Эта часть пространства имен
// определяет реализацию библиотеки

namespace cplusplus_primer {
    void inverse ( matrix &m )
        { /* ... */ }
    matrix operator+ ( const matrix &ml,   const matrix &m2 )
        { /* ... */ }
```

Например:

```
}
```

Первая часть пространства имен содержит объявления и определения, служащие интерфейсом библиотеки: определения типов, констант, объявления функций. Во второй части находятся детали реализации, то есть определения функций.

Еще более полезной для организации исходного кода библиотеки является возможность разделить определение одного пространства имен на несколько файлов: эти определения также объединяются. Наша библиотека может быть устроена следующим образом:

```
// ---- primer.h ----
namespace cplusplus_primer {
   class matrix { /*... */ };
   const double pi = 3.1416;
   matrix operator+ ( const matrix &m1, const matrix &m2 );
   void inverse( matrix & );
}

// ---- primer.C ----
#include "primer.h"
namespace cplusplus_primer {
    void inverse( matrix &m )
       { /* ... */ }
    matrix operator+ ( const matrix &m1, const matrix &m2 )
       { /* ... */ }

}
```

```
// ---- user.C ----
// определение интерфейса библиотеки
#include "primer.h"

void func( cplusplus_primer::matrix &m     )
{
   //...
   cplusplus_primer: :inverse( m );
   return m;
}
```

Программа, использующая эту библиотеку, выглядит так:

Подобная организация программы обеспечивает модульность библиотеки, необходимую для сокрытия реализации от пользователей, в то же время позволяя без ошибок скомпилировать и связать файлы `primer.C` и `user.C` в одну программу.

8.5.2. Оператор разрешения области видимости

Имя члена пользовательского пространства дополняется поставленным спереди именем этого пространства и оператором разрешения области видимости (`::`). Использование неквалифицированного члена, например `matrix`, является ошибкой. Компилятор не

```
// определение интерфейса библиотеки
#include "primer.h"

// ошибка: нет объявления для matrix
```

знает, к какому объявлению относится это имя:

```
void func( matrix &m );
```

Объявление члена пространства имен скрыто в своем пространстве. Если мы не укажем компилятору, где именно искать объявление, он произведет поиск только в текущей области видимости и в областях, включающих текущую. Допустим, если переписать предыдущую программу так:

```
// определение интерфейса библиотеки
#include "primer.h"

class matrix { /* пользовательское определение */ };

// правильно: глобальный тип matrix найден

void func( matrix &m );
```

то определение класса matrix компилятор находит в глобальной области видимости и программа компилируется без ошибок. Поскольку объявление matrix как члена пространства имен cplusplus_primer скрыто в этом пространстве, оно не конфликтует с классом, объявленным в глобальной области видимости.

Именно поэтому мы говорим, что пространства имен решают проблему засорения глобального пространства: имена их членов невидимы, если имя пространства не указано явно, с помощью оператора разрешения области видимости. Существуют и другие механизмы, позволяющие сделать объявление члена пространства имен видимым вне его. Это *using-объявления* и *using-директивы*. Мы рассмотрим их в следующем разделе.

Отметим, что оператор области видимости может быть использован и для того, чтобы сослаться на элемент глобального пространства имен. Поскольку это пространство не имеет имени, запись

```
::member_name
```

относится к его элементу. Такой способ полезен для указания членов глобального пространства, если их имена оказываются скрыты именами, объявленными во вложенных локальных областях видимости.

Следующий пример демонстрирует использование оператора области видимости для обращения к скрытому члену глобального пространства имен. Функция вычисляет последовательность чисел Фибоначчи. В программе два определения переменной max. Глобальная переменная указывает максимальное значение элемента последовательности, при превышении которого вычисление прекращается, а локальная – желаемую длину последовательности при данном вызове функции. (Напоминаем, что параметры функции относятся к ее локальной области видимости.) Внутри функции должны быть доступны обе переменных. Однако неквалифицированное имя max ссылается на локальное объявление этой переменной. Чтобы получить глобальную переменную, нужно использовать оператор разрешения области видимости ::max. Вот текст программы:

```
#include <iostream>
const int max = 65000;
const int lineLength = 12;

void fibonacci( int max )
{
    if ( max < 2 ) return;
        cout << "0 1 ";

    int v1 = 0, v2 = 1, cur;
    for ( int ix = 3; ix <= max; ++ix ) {
        cur = v1 + v2;
        if ( cur > ::max ) break;
        cout << cur << " ";
        vl = v2;
        v2 = cur;
        if (ix % "lineLength == 0) cout << end"!;
    }
}
```

```
#include <iostream>
void fibonacci( int );
int main() {
    cout << "Числа Фибоначчи: 16\n";
    fibonacci( 16 );
    return 0;
}
```

Так выглядит функция `main()`, вызывающая `fibonacci()`:

Результат работы программы:

```
Числа Фибоначчи: 16
0 1 1 2 3 5 8 13 21 34 55 89
144 233 377 610
```

8.5.3. Вложенные пространства имен

Мы уже упоминали, что пользовательские пространства имен могут быть вложенными. Такие пространства применяются для дальнейшего структурирования кода нашей библиотеки. Например:

```
// ---- primer.h ----
namespace cplusplus_primer {
    // первое вложенное пространство имен:
    // матричная часть библиотеки
    namespace MatrixLib {
        class matrix { /* ... */ };
        const double pi = 3.1416;
        matrix operators+ ( const matrix &ml, const matrix &m2 );
        void inverse( matrix & );
        // ...
    }
    // второе вложенное пространство имен:
    // зоологическая часть библиотеки
    namespace AnimalLib {
        class ZooAnimal { /* ... */ };
        class Bear : public ZooAnimal { /* ... */ };
        class Raccoon : public Bear { /* ... */ };
        // ...
    }
}
```

Пространство имен `cplusplus_primer` содержит два вложенных: `MatrixLib` и `AnimalLib`.

`cplusplus_primer` предотвращает конфликт между именами из нашей библиотеки и именами из глобального пространства вызывающей программы. Вложенность позволяет делить библиотеку на части, в которых сгруппированы связанные друг с другом объявления и определения. `MatrixLib` содержит сущности, имеющие отношение к классу `matrix`, а `AnimalLib` – к классу `ZooAnimal`.

Объявление члена вложенного пространства скрыто в этом пространстве. Имя такого члена автоматически дополняется поставленными спереди именами самого внешнего и вложенного пространств.

Например, класс, объявленный во вложенном пространстве `MatrixLib`, имеет имя

```
cplusplus_primer::MatrixLib::matrix
```

а функция

```
cplusplus_primer::MatrixLib::inverse
```

Программа, использующая члены вложенного пространства

```
#include "primer.h"

// да, это ужасно...
// скоро мы рассмотрим механизмы, облегчающие
// использование членов пространств имен!
void func( cplusplus_primer::MatrixLib::matrix &m )
{
    // ...
    cplusplus_primer::MatrixLib::inverse( m );
    return m;
```

`cplusplus_primer::MatrixLib`, выглядит так:

```
}
```

Вложенное пространство имен является вложенной областью видимости внутри пространства, содержащего его. В процессе разрешения имен вложенные пространства ведут себя так же, как вложенные блоки. Когда некоторое имя употребляется в пространстве имен, поиск его объявление проводится во всех объемлющих пространствах. В следующем примере разрешение имени Type происходит в таком порядке: сначала ищем его в пространстве имен MatrixLib, затем в cplusplus_primer и

```
typedef double Type;

namespace cplusplus_primer {
    typedef int Type; // скрывает ::Type

    namespace MatrixLib {
        int val;

        // Type: объявление найдено в cplusplus_primer
        int func(Type t) {
            double val; // скрывает MatrixLib::val
            val = ...;
        }
        // ...
    }
```

наконец в глобальной области видимости:

```
}
```

Если некоторая сущность объявляется во вложенном пространстве имен, она скрывает объявление одноименной сущности из объемлющего пространства.

В предыдущем примере имя Type из глобальной области видимости скрыто объявлением Type в пространстве cplusplus_primer. При разрешении имени Type, упоминаемого в MatrixLib, оно будет найдено в cplusplus_primer, поэтому у функции func() параметр имеет тип int.

Аналогично сущность, объявленная в пространстве имен, скрывается одноименной сущностью из вложенной локальной области видимости. В предыдущем примере имя val из MatrixLib скрыто новым объявлением val. При разрешении имени val внутри func() будет найдено его объявление в локальной области видимости, и потому присваивание в func() относится именно к локальной переменной.

8.5.4. Определение члена пространства имен

Мы видели, что определение члена пространства имен может появиться внутри определения самого пространства. Например, класс matrix и константа pi появляются внутри вложенного пространства имен MatrixLib, а определения функций operator+() и inverse() приводятся где-то в другом месте текста программы:

```
// ---- primer.h ----
namespace cplusplus_primer {
    // первое вложенное пространство имен:
    // матричная часть библиотеки
    namespace MatrixLib {
        class matrix { /* ... */ };
        const double pi = 3.1416;
        matrix operators+ ( const matrix &ml, const matrix &m2 );
        void inverse( matrix & );
        // ...
    }
}
```

Член пространства имен можно определить и вне соответствующего пространства. В таком случае имя члена должно быть квалифицировано именами пространств, к которым он принадлежит. Например, если определение функции `operator+()` помещено в

```
// ---- primer.C ----
#include "primer.h"

// определение в глобальной области видимости
cplusplus_primer::MatrixLib::matrix
    cplusplus_primer::MatrixLib::operator+
       ( const matrix& ml, const matrix &m2 )
```

глобальную область видимости, то оно должно выглядеть следующим образом:

```
           { /* ... */ }
```

Имя `operator+()` квалифицировано в данном случае именами пространств `cplusplus_primer` и `MatrixLib`. Однако обратите внимание на тип `matrix` в списке параметров `operator+()`: употреблено неквалифицированное имя. Как такое может быть?

В определении функции `operator+()` можно использовать неквалифицированные имена для членов своего пространства, поскольку определение принадлежит к его области видимости. При разрешении имен внутри функции `operator+()` используется `MatrixLib`. Заметим, однако, что в типе возвращаемого значения все же нужно указывать квалифицированное имя, поскольку он расположен вне области видимости, заданной определением функции:

```
cplusplus_primer::MatrixLib::operator+
```

В определении `operator+()` неквалифицированные имена могут встречаться в любом объявлении или выражении внутри списка параметров или тела функции. Например, локальное объявление внутри `operator+()` способно создать объект класса `matrix`:

```
// ---- primer.C ----
#include "primer.h"

cplusplus_primer::MatrixLib::matrix
    cplusplus_primer::MatrixLib::operator+
        ( const matrix &ml, const matrix &m2 )
    {
        // объявление локальной переменной типа
        // cplusplus_primer::MatrixLib::matrix
        matrix res;

        // вычислим сумму двух объектов matrix
        return res;

    }
```

Хотя члены могут быть определены вне своего пространства имен, такие определения допустимы не в любом месте. Их разрешается помещать только в пространства, объемлющие данное. Например, определение `operator+()` может появиться в глобальной области видимости, в пространстве имен `cplusplus_primer` и в

```
// ---- primer.C --
#include "primer.h"

namespace cplusplus_primer {
    MatrixLib::matrix MatrixLib::operator+
        ( const matrix &ml, const matrix &m2 ) { /* ... */ }
}
```

пространстве `MatrixLib`. В последнем случае это выглядит так:

Член может определяться вне своего пространства только при условии, что ранее он был объявлен внутри. Последнее приведенное определение `operator+()` было бы

```
namespace cplusplus_primer {
    namespace MatrixLib {
        class matrix { /*...*/ };
        // следующее объявление не может быть пропущено
        matrix operator+ ( const matrix &ml, const matrix &m2 );
        // ...
    }
}
```

ошибочным, если бы ему не предшествовало объявление в файле `primer.h`:

8.5.5. ПОО и члены пространства имен

Как уже было сказано, определение пространства имен может состоять из разрозненных частей и размещаться в разных файлах. Следовательно, член пространства разрешено объявлять во многих файлах. Например:

```
// primer.h
namespace cplusplus_primer {
    // ...
    void inverse( matrix & );
}
```

```
// usel.C
#include "primer.h"
// объявление cplusplus_primer::inverse() в use1.C
```

```
// use2.C
#include "primer.h"
// объявление cplusplus_primer::inverse() в use2.C
```

Объявление `cplusplus::inverse()` в `primer.h` ссылается на одну и ту же функцию в обоих исходных файлах `use1.C` и `use2.C`.

Член пространства имен является глобальной сущностью, хотя его имя квалифицировано. Требование ПОО (правило одного определения, см. раздел 8.2) распространяется и на него. Чтобы удовлетворить этому требованию, программы, в которых используются пространства имен, обычно организуют следующим образом:

1. Объявления функций и объектов, являющихся членами пространства имен, помещают в заголовочный файл, который включается в каждый исходный файл, где

```
// ---- primer.h ----
namespace cplusplus_primer {
    class matrix { /* ... */ };
    // объявления функций
    extern matrix operator+ ( const matrix &m1, const matrix &m2 );
    extern void inverse( matrix & );

    // объявления объектов
    extern bool error_state;
```

   они используются.

```
}
```

2. Определения этих членов помещают в исходный файл, содержащий реализацию:

```
// ---- primer.C ----
#include "primer.h"

namespace cplusplus_primer {
    // определения функций
    void inverse( matrix & )
         { /* ... */ }
    matrix operator+ ( const matrix &ml, const    matrix &m2   )
         { /" ... */ }

    // определения объектов
    bool error_state = false;
}
```

Для объявления объекта без его определения используется ключевое слово `extern`, как и в случае такого объявления в глобальной области видимости.

8.5.6. Безымянные пространства имен

Может возникнуть необходимость определить объект, функцию, класс или любую другую сущность так, чтобы она была видимой только в небольшом участке программы. Это еще один способ решения проблемы засорения глобального пространства имен. Поскольку мы уверены, что эта сущность используется ограниченно, можно не тратить время на выдумывание уникального имени. Если мы объявляем объект внутри функции или блока, его имя видимо только в этом блоке. А как сделать некоторую сущность доступной нескольким функциям, но не всей программе?

Предположим, мы хотим реализовать набор функций для сортировки вектора типа

```
// ----- SortLib.h -----
void quickSort( double *, double * );
void bubbleSort( double *, double * );
void mergeSort( double *, double * );
```

`double`:

```
void heapSort( double *, double * );
```

Все они используют одну и ту же функцию `swap()` для того, чтобы менять местами элементы вектора. Однако она не должна быть видна во всей программе, поскольку нужна только четырем названным функциям. Локализуем ее в файле `SortLib.C`.

```
// ----- SortLib.C -----
void swap( double *dl, double *d2 ) { /* ... */ }

// только эти функции используют swap()
void quickSort( double *d1, double *d2 ) { /* ... */ }
void bubbleSort( double *d1, double *d2 ) { /* ... */ }
void mergeSort( double *d1, double *d2 ) { /* ... */ }
```

Приведенный код не дает желаемого результата. Как вы думаете, почему?

```
void heapSort( double *d1, double *d2 ) { /* ... */ }
```

Хотя функция swap() определена в файле SortLib.C и не появляется в заголовочном файле SortLib.h, где содержится описание интерфейса библиотеки сортировки, она объявлена в глобальной области видимости. Следовательно, это имя является глобальным, при этом сохраняется возможность конфликта с другими именами.

Язык C++ предоставляет возможность *использования безымянного пространства имен* для объявления сущности, локальной по отношению к файлу. Определение такого пространства начинается ключевым словом namespace. Очевидно, что никакого имени за этим словом нет, а сразу же идет блок в фигурных скобках, содержащий различные

```
// ----- SortLib.C -----
namespace {
    void swap( double *dl, double *d2 ) { /* ... */ }
}
```

объявления. Например:

```
// определения функций сортировки не изменяются
```

Функция swap() видна только в файле SortLib.C. Если в другом файле в безымянном пространстве имен содержится определение swap(), то это другая функция. Наличие двух функций swap() не является ошибкой, поскольку они различны. Безымянные пространства имен отличаются от прочих: определение такого пространства локально для одного файла и не может размещаться в нескольких.

Имя swap() может употребляться в неквалифицированной форме в файле SortLib.C после определения безымянного пространства. Оператор разрешения области видимости

```
void quickSort( double *d1, double *d2 ) {
    // ...
    double* elem = d1;
    // ...
    // ссылка на член безымянного пространства имен swap()
    swap( d1, elem );
    // ...
```

для ссылки на его члены не нужен.

```
}
```

Члены безымянного пространства имен относятся к сущностям программы. Поэтому функция swap() может быть вызвана во время выполнения. Однако имена этих членов видны только внутри одного файла.

До того как в стандарте C++ появилось понятие пространства имен, наиболее удачным решением проблемы локализации было использование ключевого слова static, унаследованного из C. Член безымянного пространства имеет свойства, аналогичные глобальной сущности, объявленной как static. В языке C такая сущность невидима вне файла, в котором объявлена. Например, текст из SortLib.C можно переписать на C,

```
// SortLib.C
// swap() невидима для других файлов программы
static void swap( double *d1, double *d2 ) { /* ... */ }
```

сохранив свойства swap():

```
// определения функций сортировки такие же, как и раньше
```

Во многих программах на C++ используются объявления с ключевым словом `static`. Предполагается, что они должны быть заменены безымянными пространствами имен по мере того, как все большее число компиляторов начнет поддерживать это понятие.

Упражнение 8.11

Зачем нужно определять собственное пространство имен в программе?

Упражнение 8.12

Имеется следующее объявление `operator*()`, члена вложенного пространства имен

```
namespace cplusplus_primer {
    namespace MatrixLib {
        class matrix { /*...*/ };
        matrix operator* ( const matrix &, const matrix & );
        // ...
    }
```

`cplusplus_primer::MatrixLib`:

```
}
```

Как определить эту функцию в глобальной области видимости? Напишите только прототип.

Упражнение 8.13

Объясните, зачем нужны безымянные пространства имен.

8.6. Использование членов пространства имен A

Использование квалифицированных имен при каждом обращении к членам пространств может стать обременительным, особенно если имена пространств достаточно длинны. Если бы удалось сделать их короче, то такие имена проще было бы читать и набивать. Однако употребление коротких имен увеличивает риск их совпадения с другими, поэтому желательно, чтобы в библиотеках применялись пространства с длинными именами.

К счастью, существуют механизмы, облегчающие использование членов пространств имен в программах. *Псевдонимы пространства имен*, *using-объявления* и *using-директивы* помогают преодолеть неудобства работы с очень длинными именами.

8.6.1. Псевдонимы пространства имен

Псевдоним пространства имен используется для задания короткого синонима имени

```
namespace International_Business_Machines
```

пространства. Например, длинное имя

```
    { /* ... */ }
```

может быть ассоциировано с более коротким синонимом:

```
namespace IBM = International_Business_Machines;
```

Объявление псевдонима начинается ключевым словом namespace, за которым следует короткий псевдоним, а за ним – знак равенства и исходное полное имя пространства. Если полное имя не соответствует никакому известному пространству, это ошибка.

Псевдоним может относиться и к вложенному пространству имен. Вспомним слишком

```
#include "primer.h"

// трудно читать!
void func( cplusplus_primer::MatrixLib::matrix &m )
{
    // ...
    cplusplLis_primer::MatrixLib::inverse( m );
    return m;
```

длинное определение функции func() выше:

```
}
```

Разрешается задать псевдоним для обозначения вложенного cplusplLis_primer::MatrixLib, сделав определение функции более удобным для

```
#include "primer.h"
// более короткий псевдоним
namespace mlib = cplusplus_primer::MatrixLib;

// читать проще!
void func( mlib::matrix &m )
{
    // ...
    mlib::inverse( m );
    return m;
```

восприятия:

```
}
```

Одно пространство имен может иметь несколько взаимозаменяемых псевдонимов. Например, если псевдоним Lib ссылается на cplusplus_primer, то определение

```
// псевдоним alias относится к пространству имен cplusplus_primer
namespace alias = Lib;

void func( cplusplus_primer::matrix &m ) {
    // ...
    alias::inverse( m );
    return m;
```

функции func() может выглядеть и так:

```
}
```

8.6.2. Using-объявления

Имеется механизм, позволяющий обращаться к членам пространства имен, используя их имена без квалификатора, т.е. без префикса namespace_name::. Для этого применяются *using-объявления*.

Using-объявление начинается ключевым словом using, за которым следует

```
namespace cplusplus_primer {
    namespace MatrixLib {
         class matrix { /* ... */ };
    // ...
    }
}

// using-объявление для члена matrix
```

квалифицированное имя члена пространства. Например:

```
using cplusplus_primer::MatrixLib::matrix;
```

Using-объявление вводит имя в ту область видимости, в которой оно использовано. Так, предыдущее using-объявление делает имя matrix глобально видимым.

После того как это объявление встретилось в программе, использование имени matrix в глобальной области видимости или во вложенных в нее областях относится к этому члену пространства имен. Пусть далее идет следующее объявление:

```
void func( matrix &m );
```

Оно вводит функцию func() с параметром типа cplusplus_primer::MatrixLib::matrix.

Using-объявление ведет себя подобно любому другому объявлению: оно имеет область видимости, и имя, введенное им, можно употреблять начиная с места объявления и до конца области видимости. Using-объявление может использоваться в глобальной области видимости, равно как и в области видимости любого пространства имен. Оно употребляется и в локальной области. Имя, вводимое using-объявлением, как и любым другим, имеет следующие характеристики:

- оно должно быть уникальным в своей области видимости;
- оно скрывает одноименную сущность во внешней области;
- оно скрывается объявлением одноименной сущности во вложенной области.

Например:

```
namespace blip {
    int bi = 16, bj = 15, bk = 23;
    // прочие объявления
}
int bj = 0;

void manip() {
    using blip::bi; // bi в функции manip() ссылается на blip::bi
    ++bi;           // blip::bi == 17

    using blip::bj; // скрывает глобальную bj
                    // bj в функции manip()ссылается на blip::bj
    ++bj;           // blip::bj == 16

    int bk;         // объявление локальной bk
    using blip::bk; // ошибка: повторное определение bk в manip()
}

int wrongInit = bk; // ошибка: bk невидима

                    // надо использовать blip::bk
```

Using-объявления в функции `manip()` позволяют ссылаться на членов пространства `blib` с помощью неквалифицированных имен. Такие объявления не видны вне `manip()`, и неквалифицированные имена могут применяться только внутри этой функции. Вне ее необходимо употреблять квалифицированные имена.

Using-объявление упрощает использование членов пространства имен. Оно вводит только одно имя. Using-объявление может находиться в определенной области видимости, и, значит, мы способны точно указать, в каком месте программы те или иные члены разрешается употреблять без дополнительной квалификации.

В следующем подразделе мы расскажем, как ввести в определенную область видимости все члены некоторого пространства имен.

8.6.3. Using-директивы

Пространства имен появились в стандартном C++. Предыдущие версии C++ их не поддерживали, и, следовательно, поставляемые библиотеки не помещали глобальные объявления в пространства имен. Множество программ на C++ было написано еще до того, как компиляторы стали поддерживать такую опцию. Заключая содержимое библиотеки в пространство имен, мы можем испортить старое приложение, использующее ее предыдущие версии: все имена из этой библиотеки становятся квалифицированными, т.е. должны включать имя пространства вместе с оператором разрешения области видимости. Те приложения, в которых эти имена употребляются в неквалифицированной форме, перестают компилироваться.

Сделать видимыми имена из библиотеки, используемой в нашей программе, можно с помощью using-объявления. Предположим, что файл `primer.h` содержит интерфейс новой версии библиотеки, в котором глобальные объявления помещены в пространство имен `cplusplus_primer`. Нужно заставить нашу программу работать с новой библиотекой. Два using-объявления сделают видимыми имена класса `matrix` и функции `inverse()` из пространства `cplusplus_primer`:

```
#include "primer.h"
using cplusplus_primer::matrix;
using cplusplus_primer::inverse;

// using-объявления позволяют использовать
// имена matrix и inverse без спецификации
void func( matrix &m ) {
   // ...
   inverse( m );
   return m;
}
```

Но если библиотека достаточно велика и приложение часто использует имена из нее, то для подгонки имеющегося кода к новой библиотеке может потребоваться много using-объявлений. Добавлять их все только для того, чтобы старый код скомпилировался и заработал, утомительно и чревато ошибками. Решить эту проблему помогают *using-директивы*, облегчающие переход на новую версию библиотеки, где впервые стали применяться пространства имен.

Using-директива начинается ключевым словом using, за которым следует ключевое слово namespace, а затем имя некоторого пространства имен. Это имя должно ссылаться на определенное ранее пространство, иначе компилятор выдаст ошибку. Using-директива позволяет сделать все имена из этого пространства видимыми в неквалифицированной форме.

```
#include "pnmer.h"

// using-директива: все члены cplusplus_primer
// становятся видимыми
using namespace cplusplus_primer;

// имена matrix и inverse можно использовать без спецификации
void func( matrix &m ) {
    // ...
    inverse( m );
    return m;
}
```

Например, предыдущий фрагмент кода может быть переписан так:

Using-директива делает имена членов пространства имен видимыми за его пределами, в том месте, где она использована. Например, приведенная using-директива создает иллюзию того, что все члены cplusplus_primer объявлены в глобальной области видимости перед определением func(). При этом члены пространства имен не получают

```
namespace A {
    int i, j;
}
```

локальных псевдонимов, а как бы перемещаются в новую область видимости. Код

выглядит как

```
int i, j;
```

для фрагмента программы, содержащего в области видимости следующую using-директиву:

```
using namespace A;
```

Рассмотрим пример, позволяющий подчеркнуть разницу между using-объявлением (которое сохраняет пространство имен, но создает ассоциированные с его членами локальные синонимы) и using-директивой (которая полностью удаляет границы

```
namespace blip {
    int bi = 16, bj = 15, bk = 23;
    // прочие объявления
}
int bj = 0;

void manip() {
    using namespace blip; // using-директива -
                          // коллизия имен ::bj and blip::bj
                          // обнаруживается только при
                          // использовании bj
    ++bi;                 // blip::bi == 17
    ++bj;                 // ошибка: неоднозначность
                          // глобальная bj или blip::bj?
    ++::bj;               // правильно: глобальная bj == 1
    ++blip::bj;           // правильно: blip::bj == 16

    int bk = 97;          // локальная bk скрывает blip::bk
    ++bk;                // локальная bk == 98
}
```

пространства имен).

Во-первых, using-директивы имеют область видимости. Такая директива в функции `manip()` относится только к блоку этой функции. Для `manip()` члены пространства имен `blip` выглядят так, как будто они объявлены в глобальной области видимости, а следовательно, можно использовать их неквалифицированные имена. Вне этой функции необходимо употреблять квалифицированные.

Во-вторых, ошибки неоднозначности, вызванные применением using-директивы, обнаруживают себя при реальном обращении к такому имени, а не при встрече в тексте самой этой директивы. Например, переменная `bj`, член пространства `blib`, выглядит для `manip()` как объявленная в глобальной области видимости, вне `blip`. Однако в глобальной области уже есть такая переменная. Возникает неоднозначность имени `bj` в функции `manip()`: оно относится и к глобальной переменной, и к члену пространства `blip`. Ошибка проявляется только при упоминании `bj` в функции `manip()`. Если бы это имя вообще не использовалось в `manip()`, коллизия не проявилась бы.

В-третьих, using-директива не затрагивает употребление квалифицированных имен. Когда в `manip()` упоминается `::bj`, имеется в виду переменная из глобальной области видимости, а `blip::bj` обозначает переменную из пространства имен `blip`.

И наконец члены пространства `blip` выглядят для функции `manip()` так, как будто они объявлены в глобальной области видимости. Это означает, что локальные объявления внутри `manip()` могут скрывать имена членов пространства `blip`. Локальная

переменная bk скрывает blip::bk. Ссылка на bk внутри manip() не является неоднозначной – речь идет о локальной переменной.

Using-директивы использовать очень просто: стоит написать одну такую директиву, и все члены пространства имен сразу становятся видимыми. Однако чрезмерное увлечение ими

```
namespace cplusplus_primer {
    class matrix { };
    // прочие вещи ...
}
namespace DisneyFeatureAnimation {
    class matrix { };
    // здесь тоже ...

using namespace cplusplus_primer;
using namespace DisneyFeatureAnimation;

matrix m; //ошибка, неоднозначность:
```

возвращает нас к старой проблеме засорения глобального пространства имен:

```
// cplusplus_primer::matrix или DisneyFeatureAnimation::matrix?
```

Ошибки неоднозначности, вызываемые using-директивой, обнаруживаются только в момент использования. В данном случае – при употреблении имени matrix. Такая ошибка, найденная не сразу, может стать сюрпризом: заголовочные файлы не менялись и никаких новых объявлений в программу добавлено не было. Ошибка появилась после того, как мы решили воспользоваться новыми средствами из библиотеки.

Using-директивы очень полезны при переводе приложений на новые версии библиотек, использующие пространства имен. Однако употребление большого числа using-директив возвращает нас к проблеме засорения глобального пространства имен. Эту проблему можно свести к минимуму, если заменить using-директивы более селективными using-объявлениями. Ошибки неоднозначности, вызываемые ими, обнаруживаются в момент объявления. Мы рекомендуем пользоваться using-объявлениями, а не using-директивами, чтобы избежать засорения глобального пространства имен в своей программе.

8.6.4. Стандартное пространство имен std

Все компоненты стандартной библиотеки C++ находятся в пространстве имен std. Каждая функция, объект и шаблон класса, объявленные в стандартном заголовочном файле, таком, как <vector> или <iostream>, принадлежат к этому пространству.

Если все компоненты библиотеки объявлены в std, то какая ошибка допущена в данном примере:

```
#include <vector>
#include <string>
#include <iterator>

int main()
{
    // привязка istream_iterator к стандартному вводу
    istream_iterator<string> infile( cin );

    // istream_iterator, отмечающий end-of-stream
    istream_iterator<string> eos;

    // инициализация svec элементами, считываемыми из cin
    vector<string> svec( infile, eos );
    // ...
}
```

Правильно, этот фрагмент кода не компилируется, потому что члены пространства имен std должны использоваться с указанием их специфицированных имен. Для того чтобы исправить положение, мы можем выбрать один из следующих способов:

- заменить имена членов пространства std в этом примере соответствующими специфицированными именами;
- применить using-объявления, чтобы сделать видимыми используемые члены пространства std;
- употребить using-директиву, сделав видимыми все члены пространства std.

Членами пространства имен std в этом примере являются: шаблон класса istream_iterator, стандартный входной поток cin, класс string и шаблон класса vector.

Простейшее решение – добавить using-директиву после директивы препроцессора #include:

```
using namespace std;
```

В данном примере using-директива делает все члены пространства std видимыми. Однако не все они нам нужны. Предпочтительнее пользоваться using-объявлениями, чтобы уменьшить вероятность коллизии имен при последующем добавлении в программу глобальных объявлений.

```
using std::istream_iterator;
using std::string;
using std::cin;
```

Using-объявления, необходимые для компиляции этого примера, таковы:

```
using std::vector;
```

Но куда их поместить? Если программа состоит из большого количества файлов, можно для удобства создать заголовочный файл, содержащий все эти using-объявления, и включать его в исходные файлы вслед за заголовочными файлами стандартной библиотеки.

В нашей книге мы не употребляли using-объявлений. Это сделано, во-первых, для того, чтобы сократить размер кода, а во-вторых, потому, что большинство примеров компилировались в реализации C++, не поддерживающей пространства имен. Подразумевается, что using-объявления указаны для всех членов пространства имен `std`, используемых в примерах.

Упражнение 8.14

Поясните разницу между using-объявлениями и using-директивами.

Упражнение 8.15

Напишите все необходимые using-объявления для примера из раздела 6.14.

Упражнение 8.16

```
namespace Exercise {
    int ivar = 0;
    double dvar = 0;
    const int limit = 1000;
}
int ivar = 0;

//1
void manip() {
    //2
    double dvar = 3.1416;
    int iobj = limit + 1;
    ++ivar;
    ++::ivar;
}
```

Возьмем следующий фрагмент кода:

Каковы будут значения объявлений и выражений, если поместить using-объявления для всех членов пространства имен `Exercise` в точку `//1`? В точку `//2`? А если вместо using-объявлений использовать using-директиву?

9

9. Перегруженные функции

Итак, мы уже знаем, как объявлять, определять и использовать функции в программах. В этой главе речь пойдет об их специальном виде – перегруженных функциях. Две функции называются перегруженными, если они имеют одинаковое имя, объявлены в одной и той же области видимости, но имеют разные списки формальных параметров. Мы расскажем, как объявляются такие функции и почему они полезны. Затем мы рассмотрим вопрос об их разрешении, т.е. о том, какая именно из нескольких перегруженных функций вызывается во время выполнения программы. Эта проблема является одной из наиболее сложных в C++. Тем, кто хочет разобраться в деталях, будет интересно прочитать два раздела в конце главы, где тема преобразования типов аргументов и разрешения перегруженных функций раскрывается более подробно.

9.1. Объявления перегруженных функций

Теперь, научившись объявлять, определять и использовать функции в программах, познакомимся с *перегрузкой* – еще одним аспектом в C++. Перегрузка позволяет иметь несколько одноименных функций, выполняющих схожие операции над аргументами разных типов.

Вы уже воспользовались предопределенной перегруженной функцией. Например, для вычисления выражения

```
1 + 3
```

вызывается операция целочисленного сложения, тогда как вычисление выражения

```
1.0 + 3.0
```

осуществляет сложение с плавающей точкой. Выбор той или иной операции производится незаметно для пользователя. Операция сложения перегружена, чтобы обеспечить работу с операндами разных типов. Ответственность за распознавание контекста и применение операции, соответствующей типам операндов, возлагается на компилятор, а не на программиста.

В этой главе мы покажем, как определять собственные перегруженные функции.

9.1.1. Зачем нужно перегружать имя функции

Как и в случае со встроенной операцией сложения, нам может понадобиться набор функций, выполняющих одно и то же действие, но над параметрами различных типов. Предположим, что мы хотим определить функции, возвращающие наибольшее из переданных значений параметров. Если бы не было перегрузки, пришлось бы каждой такой функции присвоить уникальное имя. Например, семейство функций `max()` могло бы выглядеть следующим образом:

```
int i_max( int, int );
int vi_max( const vector<int> & );

int matrix_max( const matrix & );
```

Однако все они делают одно и то же: возвращают наибольшее из значений параметров. С точки зрения пользователя, здесь лишь одна операция – вычисление максимума, а детали ее реализации большого интереса не представляют.

Отмеченная лексическая сложность отражает ограничение программной среды: всякое имя, встречающееся в одной и той же области видимости, должно относиться к уникальной сущности (объекту, функции, классу и т.д.). Такое ограничение на практике создает определенные неудобства, поскольку программист должен помнить или каким-то образом отыскивать все имена. Перегрузка функций помогает справиться с этой проблемой.

Применяя перегрузку, программист может написать примерно так:

```
vector<int> vec;
//...

int ix = max( j, k );
int iy = max( vec );
```

Этот подход оказывается чрезвычайно полезным во многих ситуациях.

9.1.2. Как перегрузить имя функции

В C++ двум или более функциям может быть дано одно и то же имя при условии, что их списки параметров различаются либо числом параметров, либо их типами. В данном

```
int max ( int, int );
int max( const vector<int> & );
```

примере мы объявляем перегруженную функцию `max()`:

```
int max( const matrix & );
```

Для каждого перегруженного объявления требуется отдельное определение функции `max()` с соответствующим списком параметров.

Если в некоторой области видимости имя функции объявлено более одного раза, то второе (и последующие) объявление интерпретируется компилятором так:

- если списки параметров двух функций отличаются числом или типами

```
// перегруженные функции
void print( const string & );
```

параметров, то функции считаются перегруженными:

```
void print( vector<int> & );
```

- если тип возвращаемого значения и списки параметров в объявлениях двух

```
// объявления одной и той же функции
void print( const string &str );
```

функций одинаковы, то второе объявление считается повторным:

```
void print( const string & );
```

Имена параметров при сравнении объявлений во внимание не принимаются;

если списки параметров двух функций одинаковы, но типы возвращаемых значений различны, то второе объявление считается неправильным (несогласованным с первым) и

```
unsigned int max( int i1, int i2 );
int max( int i1, int i2 ); // ошибка: отличаются только типы
```

помечается компилятором как ошибка:

```
                           // возвращаемых значений
```

Перегруженные функции не могут различаться лишь типами возвращаемого значения;

- если списки параметров двух функций разнятся только подразумеваемыми по

```
// объявления одной и той же функции
int max ( int *ia, int sz );
```

умолчанию значениями аргументов, то второе объявление считается повторным:

```
int max ( int *ia, int = 10 );
```

Ключевое слово `typedef` создает альтернативное имя для существующего типа данных, новый тип при этом не создается. Поэтому если списки параметров двух функций различаются только тем, что в одном используется `typedef`, а в другом тип, для которого `typedef` служит псевдонимом, такие списки считаются одинаковыми, как, например, в следующих двух объявлениях функции `calc()`. В таком случае второе объявление даст ошибку компиляции, поскольку возвращаемое значение отличается от указанного

```
// typedef не вводит нового типа
typedef double DOLLAR;
// ошибка: одинаковые списки параметров, но разные типы
// возвращаемых значений
extern DOLLAR calc( DOLLAR );
```

раньше:

```
extern int calc( double );
```

Спецификаторы `const` или `volatile` при подобном сравнении не принимаются во внимание. Так, следующие два объявления считаются одинаковыми:

```
// объявляют одну и ту же функцию
void f( int );
```

```
void f( const int );
```

Спецификатор `const` важен только внутри определения функции: он показывает, что в теле функции запрещено изменять значение параметра. Однако аргумент, передаваемый по значению, можно использовать в теле функции как обычную инициированную переменную: вне функции изменения не видны. (Способы передачи аргументов, в частности передача по значению, обсуждаются в разделе 7.3.) Добавление спецификатора `const` к параметру, передаваемому по значению, не влияет на его интерпретацию. Функции, объявленной как `f(int)`, может быть передано любое значение типа `int`, равно как и функции `f(const int)`. Поскольку они обе принимают одно и то же множество значений аргумента, то приведенные объявления не считаются перегруженными. `f()` можно определить как

```
void f( int i ) { }
```

или как

```
void f( const int i ) { }
```

Наличие двух этих определений в одной программе – ошибка, так как одна и та же функция определяется дважды.

Однако, если спецификатор `const` или `volatile` применяется к параметру указательного

```
// объявляются разные функции
void f( int* );
void f( const int* );

// и здесь объявляются разные функции
void f( int& );
```

или ссылочного типа, то при сравнении объявлений он учитывается.

```
void f( const int& );
```

9.1.3. Когда не надо перегружать имя функции

В каких случаях перегрузка имени не дает преимуществ? Например, тогда, когда присвоение функциям разных имен облегчает чтение программы. Вот несколько примеров. Следующие функции оперируют одним и тем же абстрактным типом даты. На

```
void setDate( Date&, int, int, int );
Date &convertDate( const string & );
```

первый взгляд, они являются подходящими кандидатами для перегрузки:

```
void printDate( const Date& );
```

Эти функции работают с одним типом данных – классом `Date`, но выполняют семантически различные действия. В этом случае лексическая сложность, связанная с

употреблением различных имен, проистекает из принятого программистом соглашения об обеспечении набора операций над типом данных и именования функций в соответствии с семантикой этих операций. Правда, механизм классов C++ делает такое соглашение излишним. Следовало бы сделать такие функции членами класса Date, но

```
#include <string>
class Date {
public:
   set( int, int, int );
   Date& convert( const string & );
   void print();

   // ...
```

при этом оставить разные имена, отражающие смысл операции:

```
};
```

Приведем еще один пример. Следующие пять функций-членов Screen выполняют различные операции над экранным курсором, являющимся принадлежностью того же класса. Может показаться, что разумно перегрузить эти функции под общим названием

```
Screen& moveHome();
Screen& moveAbs( int, int );
Screen& moveRel( int, int, char *direction );
Screen& moveX( int );
```

move():

```
Screen& moveY( int );
```

Впрочем, последние две функции перегрузить нельзя, так как у них одинаковые списки

```
// функция, объединяющая moveX() и moveY()
```

параметров. Чтобы сделать сигнатуру уникальной, объединим их в одну функцию:

```
Screen& move( int, char xy );
```

Теперь у всех функций разные списки параметров, так что их можно перегрузить под именем move(). Однако этого делать не следует: разные имена несут информацию, без которой программу будет труднее понять. Так, выполняемые данными функциями операции перемещения курсора различны. Например, moveHome() осуществляет специальный вид перемещения в левый верхний угол экрана. Какой из двух приведенных

```
// какой вызов понятнее?
myScreen.home();          // мы считаем, что этот!
```

ниже вызовов более понятен пользователю и легче запоминается?

```
myScreen.move();
```

В некоторых случаях не нужно ни перегружать имя функции, ни назначать разные имена: применение подразумеваемых по умолчанию значений аргументов позволяет объединить несколько функций в одну. Например, функции управления курсором

```
moveAbs(int, int);
moveAbs(int, int, char*);
```

различаются наличием третьего параметра типа `char*`. Если их реализации похожи и для третьего аргумента можно найти разумное значение по умолчанию, то обе функции можно заменить одной. В данном случае на роль значения по умолчанию подойдет указатель со значением `0`:

```
move( int, int, char* = 0 );
```

Применять те или иные возможности следует тогда, когда этого требует логика приложения. Вовсе не обязательно включать перегруженные функции в программу только потому, что они существуют.

9.1.4. Перегрузка и область видимости A

Все перегруженные функции объявляются в одной и той же области видимости. К

```
#include <string>
void print( const string & );
void print( double );         // перегружает print()

void fooBar( int ival )
{
  // отдельная область видимости: скрывает обе реализации print()
  extern void print( int );

  // ошибка: print( const string & ) не видна в этой области
  print( "Value: ");
  print( ival );              // правильно: print( int ) видна
}
```

примеру, локально объявленная функция не перегружает, а просто скрывает глобальную:

Поскольку каждый класс определяет собственную область видимости, функции, являющиеся членами двух разных классов, не перегружают друг друга. (Функции-члены класса описываются в главе 13. Разрешение перегрузки для функций-членов класса рассматривается в главе 15.)

Объявлять такие функции разрешается и внутри пространства имен. С каждым из них также связана отдельная область видимости, так что функции, объявленные в разных

```
#include <string>
namespace IBM {
  extern void print( const string & );
  extern void print( double );   // перегружает print()
}
namespace Disney {
  // отдельная область видимости:
  // не перегружает функцию print() из пространства имен IBM
  extern void print( int );
```

пространствах, не перегружают друг друга. Например:

```
}
```

Использование using-объявлений и using-директив помогает сделать члены пространства имен доступными в других областях видимости. Эти механизмы оказывают определенное влияние на объявления перегруженных функций. (Using-объявления и using-директивы рассматривались в разделе 8.6.)

Каким образом using-объявление сказывается на перегрузке функций? Напомним, что оно вводит псевдоним для члена пространства имен в ту область видимости, в которой это

```
namespace libs_R_us {
  int max( int, int );
  int max( double, double );

  extern void print( int );
  extern void print( double );
}

// using-объявления
using libs_R_us::max;
using libs_R_us::print( double );   // ошибка

void func()
{
  max( 87, 65 );     // вызывает libs_R_us::max( int, int )
```

объявление встречается. Что делают такие объявления в следующей программе?

```
  max( 35.5, 76.6 ); // вызывает libs_R_us::max( double, double )
```

Первое using-объявление вводит обе функции `libs_R_us::max` в глобальную область видимости. Теперь любую из функций `max()` можно вызвать внутри `func()`. По типам аргументов определяется, какую именно функцию вызывать. Второе using-объявление – это ошибка: в нем нельзя задавать список параметров. Функция `libs_R_us::print()` объявляется только так:

```
using libs_R_us::print;
```

Using-объявление всегда делает доступными *все* перегруженные функции с указанным именем. Такое ограничение гарантирует, что интерфейс пространства имен `libs_R_us` не будет нарушен. Ясно, что в случае вызова

```
print( 88 );
```

автор пространства имен ожидает, что будет вызвана функция `libs_R_us::print(int)`. Если разрешить пользователю избирательно включать в область видимости лишь одну из нескольких перегруженных функций, то поведение программы становится непредсказуемым.

Что происходит, если using-объявление вводит в область видимости функцию с уже существующим именем? Эти функции выглядят так, как будто они объявлены прямо в том месте, где встречается using-объявление. Поэтому введенные функции участвуют в процессе разрешения имен всех перегруженных функций, присутствующих в данной области видимости:

```
#include <string>
namespace libs_R_us {
  extern void print( int );
  extern void print( double );
}

extern void print( const string & );

// libs_R_us::print( int ) и libs_R_us::print( double )
// перегружают print( const string & )
using libs_R_us::print;

void fooBar( int ival )
{
  print( "Value: ");   // вызывает глобальную функцию
                       // print( const string & )
  print( ival );       // вызывает libs_R_us::print( int )

}
```

Using-объявление добавляет в глобальную область видимости два объявления: для `print(int)` и для `print(double)`. Они являются псевдонимами в пространстве `libs_R_us` и включаются в множество перегруженных функций с именем `print`, где уже находится глобальная `print(const string &)`. При разрешении перегрузки `print` в `fooBar` рассматриваются все три функции.

Если using-объявление вводит некоторую функцию в область видимости, в которой уже имеется функция с таким же именем и таким же списком параметров, это считается ошибкой. С помощью using-объявления нельзя задать псевдоним для функции `print(int)` в пространстве имен `libs_R_us`, если в глобальной области видимости уже

```
namespace libs_R_us {
  void print( int );
  void print( double );
}

void print( int );

using libs_R_us::print; // ошибка: повторное объявление print(int)

void fooBar( int ival )
{
  print( ival );        // какая print? ::print или libs_R_us::print

}
```

есть `print(int)`. Например:

Мы показали, как связаны using-объявления и перегруженные функции. Теперь рассмотрим особенности применения using-директивы. Using-директива приводит к тому, что члены пространства имен выглядят объявленными вне этого пространства, добавляя их в новую область видимости. Если в этой области уже есть функция с тем же именем, то происходит перегрузка. Например:

```
#include <string>
namespace libs_R_us {
  extern void print( int );
  extern void print( double );
}

extern void print( const string & );

// using-директива
// print(int), print(double) и print(const string &) - элементы
// одного и того же множества перегруженных функций
using namespace libs_R_us;

void fooBar( int ival )
{
  print( "Value: ");   // вызывает глобальную функцию
                       // print( const string & )
  print( ival );       // вызывает libs_R_us::print( int )
}
```

Это верно и в том случае, когда есть несколько using-директив. Одноименные функции,

```
namespace IBM {
  int print( int );
}
namespace Disney {
  double print( double );
```

являющиеся членами разных пространств, включаются в одно и то множество:

```
// using-директива
// формируется множество перегруженных функций из различных
// пространств имен
using namespace IBM;
using namespace Disney;

long double print(long double);

int main() {
  print(1);        // вызывается IBM::print(int)
  print(3.1);      // вызывается Disney::print(double)
  return 0;
}
}
```

Множество перегруженных функций с именем `print` в глобальной области видимости включает функции `print(int)`, `print(double)` и `print(long double)`. Все они рассматриваются в `main()` при разрешении перегрузки, хотя первоначально были определены в разных пространствах имен.

Итак, повторим, что перегруженные функции находятся в одной и той же области видимости. В частности, они оказываются там в результате применения using-объявлений и using-директив, делающих доступными имена из других областей.

9.1.5. Директива extern "C" и перегруженные функции A

В разделе 7.7 мы видели, что директиву связывания `extern "C"` можно использовать в программе на C++ для того, чтобы указать, что некоторый объект находится в части, написанной на языке C. Как эта директива влияет на объявления перегруженных функций? Могут ли в одном и том же множестве находиться функции, написанные как на C++, так и на C?

В директиве связывания разрешается задать только одну из множества перегруженных

```
// ошибка: для двух перегруженных функций указана директива extern "C"
extern "C" void print( const char* );
```

функций. Например, следующая программа некорректна:

```
extern "C" void print( int );
```

Приведенный ниже пример перегруженной функции `calc()` иллюстрирует типичное

```
class SmallInt ( /* ... */ );
class BigNum ( /* ... */ );

// написанная на C функция может быть вызвана как из программы,
// написанной на C, так и из программы, написанной на C++.
// функции C++ обрабатывают параметры, являющиеся классами
extern "C" double calc( double );
extern SmallInt calc( const SmallInt& );
```

применение директивы `extern "C"`:

```
extern BigNum calc( const BigNum& );
```

Написанная на C функция `calc()` может быть вызвана как из C, так и из программы на C++. Остальные две функции принимают в качестве параметра класс и, следовательно, их допустимо использовать только в программе на C++. Порядок следования объявлений несуществен.

Директива связывания не имеет значения при решении, какую функцию вызывать; важны только типы параметров. Выбирается та функция, которая лучше всего

```
Smallint si = 8;
int main() {
   calc( 34 );     // вызывается C-функция calc( double )
   calc( si );     // вызывается функция C++ calc( const SmallInt & )
   // ...
   return 0;
```

соответствует типам переданных аргументов:

```
}
```

9.1.6. Указатели на перегруженные функции A

Можно объявить указатель на одну из множества перегруженных функций. Например:

```
extern void ff( vector<double> );
extern void ff( unsigned int );

// на какую функцию указывает pf1?
void ( *pf1 )( unsigned int ) = &ff;
```

Поскольку функция `ff()` перегружена, одного инициализатора `&ff` недостаточно для выбора правильного варианта. Чтобы понять, какая именно функция инициализирует указатель, компилятор ищет в множестве всех перегруженных функций ту, которая имеет тот же тип возвращаемого значения и список параметров, что и функция, на которую ссылается указатель. В нашем случае будет выбрана функция `ff(unsigned int)`.

А что если не найдется функции, в точности соответствующей типу указателя? Тогда

```
extern void ff( vector<double> );
extern void ff( unsigned int );

// ошибка: соответствие не найдено: неверный список параметров
```

компилятор выдаст сообщение об ошибке:

```
// ошибка: соответствие не найдено: неверный тип возвращаемого значения
void ( *pf2 )( int ) = &ff;
double ( *pf3 )( vector<double> ) = &ff;
```

Присваивание работает аналогично. Если значением указателя должен стать адрес перегруженной функции , то для выбора операнда в правой части оператора присваивания используется тип указателя на функцию. И если компилятор не находит функции, в точности соответствующей нужному типу, он выдает сообщение об ошибке. Таким образом, преобразование типов между указателями на функции никогда не

```
matrix calc( const matrix & );
int calc( int, int );

int ( *pc1 )( int, int ) = 0;
int ( *pc2 )( int, double ) = 0;

// ...
// правильно: выбирается функция calc( int, int )
pc1 = &calc;

// ошибка: нет соответствия: неверный тип второго параметра
```

производится.

```
pc2 = &calc;
```

9.1.7. Безопасное связывание A

При использовании перегрузки складывается впечатление, что в программе можно иметь несколько одноименных функций с разными списками параметров. Однако это лексическое удобство существует только на уровне исходного текста. В большинстве систем компиляции программы, обрабатывающие этот текст для получения исполняемого

кода, требуют, чтобы все имена были различны. Редакторы связей, как правило, разрешают внешние ссылки лексически. Если такой редактор встречает имя `print` два или более раз, он не может различить их путем анализа типов (к этому моменту информация о типах обычно уже потеряна). Поэтому он просто печатает сообщение о повторно определенном символе `print` и завершает работу.

Чтобы разрешить эту проблему, имя функции вместе с ее списком параметров *декорируется* так, чтобы получилось уникальное внутреннее имя. Вызываемые после компилятора программы видят только это внутреннее имя. Как именно производится такое преобразование имен, зависит от реализации. Общая идея заключается в том, чтобы представить число и типы параметров в виде строки символов и дописать ее к имени функции.

Как было сказано в разделе 8.2, такое кодирование гарантирует, в частности, что два объявления одноименных функций с разными списками параметров, находящиеся в разных файлах, не воспринимаются редактором связей как объявления одной и той же функции. Поскольку этот способ помогает различить перегруженные функции на фазе редактирования связей, мы говорим о *безопасном связывании*.

Декорирование имен не применяется к функциям, объявленным с помощью директивы `extern "C"`, так как лишь одна из множества перегруженных функций может быть написана на чистом C. Две функции с различными списками параметров, объявленные как `extern "C"`, редактор связей воспринимает как один и тот же символ.

Упражнение 9.1

Зачем может понадобиться объявлять перегруженные функции?

Упражнение 9.2

Как нужно объявить перегруженные варианты функции `error()`, чтобы были корректны

```
int index;
int upperBound;
char selectVal;
// ...
error( "Array out of bounds: ", index, upperBound );
error( "Division by zero" );
```

следующие вызовы:

```
error( "Invalid selection", selectVal );
```

Упражнение 9.3

Объясните, к какому эффекту приводит второе объявление в каждом из приведенных примеров:

```
(a) int calc( int, int );
    int calc( const int, const int );

(b) int get();
    double get();

(c) int *reset( int * );
    double *reset( double * ):

(d) extern "C" int compute( int *, int );

    extern "C" double compute( double *, double );
```

Упражнение 9.4

```
(a) void reset( int * );
    void (*pf)( void * ) = reset;

(b) int calc( int, int );
    int (*pf1)( int, int ) = calc;

(c) extern "C" int compute( int *, int );
    int (*pf3)( int*, int ) = compute;
```

Какая из следующих инициализаций приводит к ошибке? Почему?

```
 (d) void (*pf4)( const matrix & ) = 0;
```

9.2. Три шага разрешения перегрузки

Разрешением перегрузки функции называется процесс выбора той функции из множества перегруженных, которую следует вызвать. Этот процесс основывается на указанных при

```
T t1, t2;
void f( int, int );
void f( float, float );

int main() {
   f( t1, t2 );
   return 0;
}
```

вызове аргументах. Рассмотрим пример:

Здесь в ходе процесса разрешения перегрузки в зависимости от типа `T` определяется, будет ли при обработке выражения `f(t1,t2)` вызвана функция `f(int,int)` или `f(float,float)` или зафиксируется ошибка.

Разрешение перегрузки функции – один и самых сложных аспектов языка C++. Пытаясь разобраться во всех деталях, начинающие программисты столкнутся с серьезными трудностями. Поэтому в данном разделе мы представим лишь краткий обзор того, как происходит разрешение перегрузки, чтобы у вас составилось хоть какое-то впечатление

об этом процессе. Для тех, кто хочет узнать больше, в следующих двух разделах приводится более подробное описание.

Процесс разрешения перегрузки функции состоит из трех шагов, которые мы покажем на

```
void f();
void f( int );
void f( double, double = 3.4 );
void f( char *, char * );

void main() {
   f( 5.6 );
   return 0;
}
```

следующем примере:

При разрешении перегрузки функции выполняются следующие шаги:

1. Выделяется множество перегруженных функций для данного вызова, а также свойства списка аргументов, переданных функции.
2. Выбираются те из перегруженных функций, которые могут быть вызваны с данными аргументами, с учетом их количества и типов.
3. Находится функция, которая лучше всего соответствует вызову.

Рассмотрим последовательно каждый пункт.

На первом шаге необходимо идентифицировать множество перегруженных функций, которые будут рассматриваться при данном вызове. Вошедшие в это множество функции называются *кандидатами*. Функция-кандидат – это функция с тем же именем, что и вызванная, причем ее объявление видимо в точке вызова. В нашем примере есть четыре таких кандидата: `f()`, `f(int)`, `f(double, double)` и `f(char*, char*)`.

После этого идентифицируются свойства списка переданных аргументов, т.е. их количество и типы. В нашем примере список состоит из двух аргументов типа `double`.

На втором шаге среди множества кандидатов отбираются *устоявшие* (viable) – такие, которые могут быть вызваны с данными аргументами, Устоявшая функция либо имеет столько же формальных параметров, сколько фактических аргументов передано вызванной функции, либо больше, но тогда для каждого дополнительного параметра должно быть задано значение по умолчанию. Чтобы функция считалась устоявшей, для любого фактического аргумента, переданного при вызове, обязано существовать преобразование к типу формального параметра, указанного в объявлении.

В нашем примере есть две устоявших функции, которые могут быть вызваны с приведенными аргументами:

- функция `f(int)` устояла, потому что у нее есть всего один параметр и существует преобразование фактического аргумента типа `double` к формальному параметру типа `int`;
- функция `f(double,double)` устояла, потому что для второго аргумента есть значение по умолчанию, а первый формальный параметр имеет тип `double`, что в точности соответствует типу фактического аргумента.

Если после второго шага не нашлось устоявших функций, то вызов считается ошибочным. В таких случаях мы говорим, что имеет место *отсутствие соответствия*.

Третий шаг заключается в выборе функции, лучше всего отвечающей контексту вызова. Такая функция называется *наилучшей из устоявших* (или *наиболее подходящей)*. На этом шаге производится *ранжирование* преобразований, использованных для приведения типов фактических аргументов к типам формальных параметров устоявшей функции. Наиболее подходящей считается функция, для которой выполняются следующие условия:

преобразования, примененные к фактическим аргументам, *не хуже* преобразований, необходимых для вызова любой другой устоявшей функции;

для некоторых аргументов примененные преобразования *лучше*, чем преобразования, необходимые для приведения тех же аргументов в вызове других устоявших функций.

Преобразования типов и их ранжирование более подробно обсуждаются в разделе 9.3. Здесь мы лишь кратко рассмотрим ранжирование преобразований для нашего примера. Для устоявшей функции `f(int)` должно быть применено приведение фактического аргумента типа `double` к типу `int`, относящееся к числу стандартных. Для устоявшей функции `f(double,double)` тип фактического аргумента `double` в точности соответствует типу формального параметра. Поскольку точное соответствие лучше стандартного преобразования (отсутствие преобразования всегда лучше, чем его наличие), то наиболее подходящей функцией для данного вызова считается `f(double,double)`.

Если на третьем шаге не удается отыскать единственную лучшую из устоявших функцию, иными словами, нет такой устоявшей функции, которая подходила бы больше всех остальных, то вызов считается *неоднозначным*, т.е. ошибочным.

(Более подробно все шаги разрешения перегрузки функции обсуждаются в разделе 9.4. Процесс разрешения используется также при вызовах перегруженной функции-члена класса и перегруженного оператора. В разделе 15.10 рассматриваются правила разрешения перегрузки, применяемые к функциям-членам класса, а в разделе 15.11 – правила для перегруженных операторов. При разрешении перегрузки следует также принимать во внимание функции, конкретизированные из шаблонов. В разделе 10.8 обсуждается, как шаблоны влияют на такое разрешение.)

Упражнение 9.5

Что происходит на последнем (третьем) шаге процесса разрешения перегрузки функции?

9.3. Преобразования типов аргументов A

На втором шаге процесса разрешения перегрузки функции компилятор идентифицирует и ранжирует преобразования, которые следует применить к каждому фактическому аргументу вызванной функции для приведения его к типу соответствующего формального параметра любой из устоявших функций. Ранжирование может дать один из трех возможных результатов:

- *точное соответствие*. Тип фактического аргумента точно соответствует типу формального параметра. Например, если в множестве перегруженных функций

```
void print( unsigned int );
void print( const char* );
```

`print()` есть такие:

```
void print( char );
```

```
unsigned int a;

print( 'a' );    // соответствует print( char );
print( "a" );    // соответствует print( const char* );
```

то каждый из следующих трех вызовов дает точное соответствие:

```
print( a );      // соответствует print( unsigned int );
```

- *соответствие с преобразованием типа*. Тип фактического аргумента не

```
void ff( char );
```

соответствует типу формального параметра, но может быть преобразован в него:

```
ff( 0 );         // аргумент типа int приводится к типу char
```

- *отсутствие соответствия*. Тип фактического аргумента не может быть приведен к типу формального параметра в объявлении функции, поскольку необходимого преобразования не существует. Для каждого из следующих двух

```
// функции print() объявлены так же, как и выше
int *ip;
class SmallInt { /* ... */ };
SmallInt si;

print( ip );     // ошибка: нет соответствия
```

вызовов функции `print()` соответствия нет:

```
print( si );     // ошибка: нет соответствия
```

Для установления точного соответствия тип фактического аргумента необязательно должен совпадать с типом формального параметра. К аргументу могут быть применены некоторые тривиальные преобразования, а именно:

- преобразование l-значения в r-значение;
- преобразование массива в указатель;
- преобразование функции в указатель;
- преобразования спецификаторов.

(Подробнее они рассмотрены ниже.)

Категория соответствия с преобразованием типа является наиболее сложной. Необходимо рассмотреть несколько видов такого приведения: *расширение типов* (promotions), *стандартные преобразования* и *определенные пользователем преобразования*. (Расширения типов и стандартные преобразования изучаются в этой главе. Определенные пользователем преобразования будут представлены позднее, после детального рассмотрения классов; они выполняются *конвертером*, функцией-членом, которая позволяет определить в классе собственный набор “стандартных” трансформаций. В главе 15 мы познакомимся с такими конвертерами и с тем, как они влияют на разрешение перегрузки функций.)

При выборе лучшей из устоявших функций для данного вызова компилятор ищет функцию, для которой применяемые к фактическим аргументам преобразования являются “наилучшими”. Преобразования типов ранжируются следующим образом: точное соответствие лучше расширения типа, расширение типа лучше стандартного преобразования, а оно, в свою очередь, лучше определенного пользователем преобразования. Мы еще вернемся к ранжированию в разделе 9.4, а пока на простых примерах покажем, как оно помогает выбрать наиболее подходящую функцию.

9.3.1. Подробнее о точном соответствии

Самый простой случай возникает тогда, когда типы фактических аргументов совпадают с типами формальных параметров. Например, есть две показанные ниже перегруженные функции `max()`. Тогда каждый из вызовов `max()` точно соответствует одному из

```
int max( int, int );
double max( double, double );

int i1;

void calc( double d1 ) {
   max( 56, i1 );    // точно соответствует max( int, int );
   max( d1, 66.9 );  // точно соответствует max( double, double );
}
```

объявлений:

Перечислимый тип точно соответствует только определенным в нем элементам

```
enum Tokens { INLINE = 128; VIRTUAL = 129; };
Tokens curTok = INLINE;

enum Stat { Fail, Pass };

extern void ff( Tokens );
extern void ff( Stat );
extern void ff( int );

int main() {
   ff( Pass );       // точно соответствует ff( Stat )
   ff( 0 );          // точно соответствует ff( int )
   ff( curTok );     // точно соответствует ff( Tokens )
   // ...
}
```

перечисления, а также объектам, которые объявлены как принадлежащие к этому типу:

Выше уже упоминалось, что фактический аргумент может точно соответствовать формальному параметру, даже если для приведения их типов необходимо некоторое тривиальное преобразование, первое из которых – преобразование l-значения в r-значение. Под l-значением понимается объект, удовлетворяющий следующим условиям:

- можно получить адрес объекта;
- можно получить значение объекта;

- это значение легко модифицировать (если только в объявлении объекта нет спецификатора const).

Напротив, r-значение – это выражение, значение которого вычисляется, или выражение, обозначающее временный объект, для которого нельзя получить адрес и значение

```
int calc( int );

int main() {
   int lval, res;

   lval = 5;     // lvalue: lval; rvalue: 5
   res = calc( lval );
                 // lvalue: res
                 // rvalue: временный объект для хранения значения,
                 // возвращаемого функцией calc()
   return 0;
}
```

которого нельзя модифицировать. Вот простой пример:

В первом операторе присваивания переменная lval – это l-значение, а литерал 5 – r-значение. Во втором операторе присваивания res – это l-значение, а временный объект, в котором хранится результат, возвращаемый функцией calc(), – это r-значение.

В некоторых ситуациях в контексте, где ожидается значение, можно использовать

```
int obj1;
int obj2;

int main() {
   // ...
   int local = obj1 + obj2;
   return 0;
}
```

выражение, представляющее собой l-значение:

Здесь obj1 и obj2 – это l-значения. Однако для выполнения сложения в функции main() из переменных obj1 и obj2 извлекаются их значения. Действие, состоящее в извлечении значения объекта, представленного выражением вида l-значение, называется преобразованием l-значения в r-значение.

Когда функция ожидает аргумент, переданный по значению, то в случае, если аргумент

```
#include <string>
string color( "purple" );
void print( string );

int main() {
   print( color );    // точное соответствие: преобразование lvalue
                      // в rvalue
   return 0;
}
```

является l-значением, выполняется его преобразование в r-значение:

Так как аргумент в вызове `print(color)` передается по значению, то производится преобразование l-значения в r-значение для извлечения значения `color` и передачи его в функцию с прототипом `print(string)`. Однако несмотря на то, что такое приведение имело место, считается, что фактический аргумент `color` точно соответствует объявлению `print(string)`.

При вызове функций не всегда требуется применять к аргументам подобное преобразование. Ссылка представляет собой l-значение; если у функции есть параметр-ссылка, то при вызове функция получает l-значение. Поэтому к фактическому аргументу, которому соответствует формальный параметр-ссылка, описанное преобразование не

```
#include <list>
```

применяется. Например, пусть объявлена такая функция:

```
void print( list<int> & );
```

В вызове ниже `li` – это l-значение, представляющее объект `list<int>`, передаваемый

```
list<int> li(20);

int main() {
   // ...
   print( li );     // точное соответствие: нет преобразования lvalue в
                    // rvalue
   return 0;
```

функции `print()`:

```
}
```

Сопоставление `li` с параметром-ссылкой считается точным соответствием.

Второе преобразование, при котором все же фиксируется точное соответствие, – это преобразование массива в указатель. Как уже отмечалось в разделе 7.3, параметр функции никогда не имеет тип массива, трансформируясь вместо этого в указатель на его первый элемент. Аналогично фактический аргумент типа массива из NT (где N – число элементов в массиве, а T – тип каждого элемента) всегда приводится к типу указателя на T. Такое преобразование типа фактического аргумента и называется преобразованием массива в указатель. Несмотря на это, считается, что фактический аргумент точно

```
int ai[3];
void putValues(int *);

int main() {
   // ...
   putValues(ai);   // точное соответствие: преобразование массива в
                    // указатель
   return 0;
```

соответствует формальному параметру типа “указатель на T”. Например:

```
}
```

Перед вызовом функции `putValues()` массив преобразуется в указатель, в результате чего фактический аргумент `ai` (массив из трех целых) приводится к указателю на `int`.

Хотя формальным параметром функции putValues() является указатель и фактический аргумент при вызове преобразован, между ними устанавливается точное соответствие.

При установлении точного соответствия допустимо также преобразование функции в указатель. (Оно упоминалось в разделе 7.9.) Как и параметр-массив, параметр-функция становится указателем на функцию. Фактический аргумент типа “функция” также автоматически приводится к типу указателя на функцию. Такое преобразование типа фактического аргумента и называется преобразованием функции в указатель. Хотя трансформация производится, считается, что фактический аргумент точно соответствует

```
int lexicoCompare( const string &, const string & );

typedef int (*PFI)( const string &, const string & );
void sort( string *, string *, PFI );

string as[10];

int main()
{
   // ...
   sort( as,
         as + sizeof(as)/sizeof(as[0] - 1 ),
         lexicoCompare   // точное соответствие
                         // преобразование функции в указатель
        );

   return 0;
}
```

формальному параметру. Например:

Перед вызовом sort() применяется преобразование функции в указатель, которое приводит аргумент lexicoCompare от типа “функция” к типу “указатель на функцию”. Хотя формальным параметром функции является указатель, а фактическим – имя функции и, следовательно, было произведено преобразование функции в указатель, считается, что фактический аргумент точно третьему формальному параметру функции sort().

Последнее из перечисленных выше – это преобразование спецификаторов. Оно относится только к указателям и заключается в добавлении спецификаторов const или volatile

```
int a[5] = { 4454, 7864, 92, 421, 938 };
int *pi = a;
bool is_equal( const int * , const int * );

void func( int *parm ) {

   // точное соответствие между pi и parm: преобразование спецификаторов
   if ( is_equal( pi, parm ) )
      // ...

     return 0;
}
```

(или обоих) к типу, который адресует данный указатель:

Перед вызовом функции `is_equal()` фактические аргументы `pi` и `parm` преобразуются из типа “указатель на `int`” в тип “указатель на `const int`”. Эта трансформация заключается в добавлении спецификатора `const` к адресуемому типу, поэтому относится к категории преобразований спецификаторов. Несмотря на то, что функция ожидает получить два указателя на `const int`, а фактические аргументы являются указателями на `int`, считается, что точное соответствие между формальными и фактическими параметрами функции `is_equal()` установлено.

Преобразование спецификаторов применимо только к типу, который адресует указатель. Оно не употребляется в случае, когда формальный параметр имеет спецификатор `const`

```
extern void takeCI( const int );

int main() {
   int ii = ...;
   takeCI(ii);     // преобразование спецификаторов не применяется
   return 0;
}
```

или `volatile`, а фактический аргумент – нет.

Хотя формальный параметр функции `takeCI()` имеет тип `const int`, а вызывается она с аргументом `ii` типа `int`, преобразование спецификаторов не производится: есть точное соответствие между фактическим аргументом и формальным параметром.

Все сказанное верно и для случая, когда аргумент является указателем, а спецификаторы

```
extern void init( int *const );
extern int *pi;

int main() {
   // ...
   init(pi);     // преобразование спецификаторов не применяется
   return 0;
}
```

`const` или `volatile` относятся к этому указателю:

Спецификатор `const` при формальном параметре функции `init()` относится к самому указателю, а не к типу, который он адресует. Поэтому компилятор при анализе преобразований, которые должны быть применены к фактическому аргументу, не учитывает этот спецификатор. К аргументу `pi` не применяется преобразование спецификатора: считается, что этот аргумент и формальный параметр точно соответствуют друг другу.

Первые три из рассмотренных преобразований (l-значения в r-значение, массива в указатель и функции в указатель) часто называют *трансформациями l-значений*. (В разделе 9.4 мы увидим, что хотя и трансформации l-значений, и преобразования спецификаторов относятся к категории преобразований, не нарушающих точного соответствия, его степень считается выше в случае, когда необходима лишь первая трансформация. В следующем разделе мы поговорим об этом несколько подробнее.)

Точное соответствие можно установить принудительно, воспользовавшись явным приведением типов. Например, если есть две перегруженные функции:

```
extern void ff(int);
extern void ff(void *);
```

то вызов

```
ff( 0xffbc );   // вызывается ff(int)
```

будет точно соответствовать `ff(int)`, хотя литерал `0xffbc` записан в виде шестнадцатеричной константы. Программист может заставить компилятор вызвать функцию `ff(void *)`, если явно выполнит операцию приведения типа:

```
ff( reinterpret_cast<void *>(0xffbc) );    // вызывается ff(void*)
```

Если к фактическому аргументу применяется такое приведение, то он приобретает тип, в который преобразуется. Явные приведения типов помогают в управлении процессом разрешения перегрузки. Например, если при разрешении перегрузки получается неоднозначный результат (фактические аргументы одинаково хорошо соответствуют двум или более устоявшим функциям), то для устранения неоднозначности можно применить явное приведение типа, заставив компилятор выбрать конкретную функцию.

9.3.2. Подробнее о расширении типов

Под расширением типа понимается одно из следующих преобразований:

- фактический аргумент типа `char`, `unsigned char` или `short` расширяется до типа `int`. Фактический аргумент типа `unsigned short` расширяется до типа `int`, если машинный размер `int` больше, чем размер `short`, и до типа `unsigned int` в противном случае;
- аргумент типа `float` расширяется до типа `double`;
- аргумент перечислимого типа расширяется до первого из следующих типов, который способен представить все значения элементов перечисления: `int`, `unsigned int`, `long`, `unsigned long`;
- аргумент типа `bool` расширяется до типа `int`.

Подобное расширение применяется, когда тип фактического аргумента совпадает с одним из только что перечисленных типов, а формальный параметр относится к

```
extern void manip( int );

int main() {
   manip( 'a' );    // тип char расширяется до int
   return 0;
}
```

соответствующему расширенному типу:

Символьный литерал имеет тип `char`. Он расширяется до `int`. Поскольку расширенный тип соответствует типу формального параметра функции `manip()`, мы говорим, что ее вызов требует расширения типа аргумента.

```
extern void print( unsigned int );
extern void print( int );
extern void print( char );

unsigned char uc;
```

Рассмотрим следующий пример:

```
print( uc );   // print( int ); для uc требуется только расширение типа
```

Для аппаратной платформы, на которой unsigned char занимает один байт памяти, а int – четыре байта, расширение преобразует unsigned char в int, так как с его помощью можно представить все значения типа unsigned char. Для такой машинной архитектуры из приведенного в примере множества перегруженных функций наилучшее соответствие аргументу типа unsigned char обеспечивает print(int). Для двух других функций установление соответствия требует стандартного приведения.

Следующий пример иллюстрирует расширение фактического аргумента перечислимого

```
enum Stat ( Fail, Pass );

extern void ff( int );
extern void ff( char );

int main() {
   // правильно: элемент перечисления Pass расширяется до типа int
   ff( Pass );    // ff( int )
   ff( 0 );       // ff( int )
```

типа:

```
}
```

Иногда расширение перечислений преподносит сюрпризы. Компиляторы часто выбирают представление перечисления в зависимости от значений его элементов. Предположим, что в вышеупомянутой архитектуре (один байт для char и четыре байта для int) определено такое перечисление:

```
enum e1 { a1, b1, c1 };
```

Поскольку есть всего три элемента: a1, b1 и c1 со значениями 0, 1 и 2 соответственно – и поскольку все эти значения можно представить типом char, то компилятор, как правило, и выбирает char для представления типа e1. Рассмотрим, однако, перечисление e2 со следующим множеством элементов:

```
enum e2 { a2, b2, c2=0x80000000 };
```

Так как одна из констант имеет значение 0x80000000, то компилятор обязан выбрать для представления e2 такой тип, который достаточен для хранения значения 0x80000000, то есть unsigned int.

Итак, хотя и e1, и e2 являются перечислениями, их представления различаются. Из-за этого e1 и e2 расширяются до разных типов:

```
#include <string>

string format( int );
string format( unsigned int );

int main() {
   format(a1);    // вызывается format( int )
   format(a2);    // вызывается format( unsigned int )
   return 0;
}
```

При первом обращении к `format()` фактический аргумент расширяется до типа `int`, так как для представления типа `e1` используется `char`, и, следовательно, вызывается перегруженная функция `format(int)`. При втором обращении тип фактического аргумента `e2` представлен типом `unsigned int` и аргумент расширяется до `unsigned int`, из-за чего вызывается перегруженная функция `format(unsigned int)`. Поэтому следует помнить, что поведение двух перечислений по отношению к процессу разрешения перегрузки может быть различным и зависеть от значений элементов, определяющих, как происходит расширение типа.

9.3.3. Подробнее о стандартном преобразовании

Имеется пять видов стандартных преобразований, а именно:

1. преобразования целых типов: приведение от целого типа или перечисления к любому другому целому типу (исключая трансформации, которые выше были отнесены к категории расширения типов);
2. преобразования типов с плавающей точкой: приведение от любого типа с плавающей точкой к любому другому типу с плавающей точкой (исключая трансформации, которые выше были отнесены к категории расширения типов);
3. преобразования между целым типом и типом с плавающей точкой: приведение от любого типа с плавающей точкой к любому целому типу или наоборот;
4. преобразования указателей: приведение целого значения `0` к типу указателя или трансформация указателя любого типа в тип `void*`;
5. преобразования в тип `bool`: приведение от любого целого типа, типа с плавающей точкой, перечислимого типа или указательного типа к типу `bool`.

```
extern void print( void* );
extern void print( double );

int main() {
   int i;
   print( i );   // соответствует print( double );
                 // i подвергается стандартному преобразованию из int в
          double
   print( &i );  // соответствует print( void* );
                 // &i подвергается стандартному преобразованию
                 // из int* в void*
   return 0;
```

Вот несколько примеров:

```
}
```

Преобразования, относящиеся к группам 1, 2 и 3, потенциально опасны, так как целевой тип может и не обеспечивать представления всех значений исходного. Например, с помощью float нельзя адекватно представить все значения типа int. Именно по этой причине трансформации, входящие в эти группы, отнесены к категории стандартных

```
int i;
void calc( float );
int main() {
   calc( i );   // стандартное преобразование между целым типом и типом с
                // плавающей точкой потенциально опасно в зависимости от
                // значения i
   return 0;
```

преобразований, а не расширений типов.

```
}
```

При вызове функции calc() применяется стандартное преобразование из целого типа int в тип с плавающей точкой float. В зависимости от значения переменной i может оказаться, что его нельзя сохранить в типе float без потери точности.

Предполагается, что все стандартные изменения требуют одного объема работы. Например, преобразование из char в unsigned char не более приоритетно, чем из char в double. Близость типов не принимается во внимание. Если две устоявших функции требуют для установления соответствия стандартной трансформации фактического аргумента, то вызов считается неоднозначным и помечается компилятором как ошибка.

```
extern void manip( long );
```

Например, если даны две перегруженные функции:

```
extern void manip( float );
```

```
int main() {
   manip( 3.14 );   // ошибка: неоднозначность
                    // manip( float ) не лучше, чем manip( int )
   return 0;
```

то следующий вызов неоднозначен:

```
}
```

Константа 3.14 имеет тип double. С помощью того или иного стандартного преобразования соответствие может быть установлено с любой из перегруженных функций. Поскольку есть две трансформации, приводящие к цели, вызов считается неоднозначным. Ни одно преобразование не имеет преимущества над другим. Программист может разрешить неоднозначность либо путем явного приведения типа:

```
manip ( static_cast<long>( 3.14 ) );   // manip( long )
```

либо используя суффикс, обозначающий, что константа принадлежит к типу float:

```
manip ( 3.14F ) );   // manip( float )
```

Вот еще несколько примеров неоднозначных вызовов, которые помечаются как ошибки,

```
extern void farith( unsigned int );
extern void farith( float );

int main() {
   // каждый из последующих вызовов неоднозначен
   farith( 'a' );       // аргумент имеет тип char
   farith( 0 );         // аргумент имеет тип int
   farith( 2uL );       // аргумент имеет тип unsigned long
   farith( 3.14159 );   // аргумент имеет тип double
   farith( true );      // аргумент имеет тип bool
```

поскольку соответствуют нескольким перегруженным функциям:

```
}
```

Стандартные преобразования указателей иногда противоречат интуиции. В частности, значение `0` приводится к указателю на любой тип; полученный таким образом указатель называется *нулевым*. Значение `0` может быть представлено как константное выражение

```
void set(int*);

int main() {
   // преобразование указателя из 0 в int* применяется к аргументам
   // в обоих вызовах
   set( 0L );
   set( 0x00 );
   return 0;
```

целого типа:

```
}
```

Константное выражение `0L` (значение `0` типа `long int`) и константное выражение `0x00` (шестнадцатеричное целое значение `0`) имеют целый тип и потому могут быть преобразованы в нулевой указатель типа `int*`.

Но поскольку перечисления не относятся к целым типам, элемент, равный `0`, не приводим

```
enum EN { zr = 0 };
```

к типу указателя:

```
set( zr );   // ошибка: zr нельзя преобразовать в тип int*
```

Вызов функции `set()` является ошибкой, так как не существует преобразования между значением `zr` элемента перечисления и формальным параметром типа `int*`, хотя `zr` равно `0`.

Следует отметить, что константное выражение `0` имеет тип `int`. Для его приведения к типу указателя требуется стандартное преобразование. Если в множестве перегруженных функций есть функция с формальным параметром типа `int`, то именно в ее пользу будет разрешена перегрузка в случае, когда фактический аргумент равен `0`:

```
void print( int );
void print( void * );

void set( const char * );
void set( char * );

int main () {
   print( 0 );   // вызывается print( int );
   set( 0 );     // неоднозначность
   return 0;
}
```

При вызове print(int) имеет место точное соответствие, тогда как для вызова print(void*) необходимо приведение значения 0 к типу указателя. Поскольку соответствие лучше преобразования, для разрешения этого вызова выбирается функция print(int). Обращение к set() неоднозначно, так как 0 соответствует формальным параметрам обеих перегруженных функций за счет применения стандартной трансформации. Раз обе функции одинаково хороши, фиксируется неоднозначность.

Последнее из возможных преобразований указателя позволяет привести указатель любого типа к типу void*, поскольку void* – это родовой указатель на любой тип данных. Вот

```
#include <string>
extern void reset( void * );

void func( int *pi, string *ps ) {
   // ...
   reset( pi );   // преобразование указателя: int* в void*
   /// ...
   reset( ps );   // преобразование указателя: string* в void*
}
```

несколько примеров:

Только указатели на типы данных могут быть приведены к типу void* с помощью

```
typedef int (*PFV)();
extern PFV testCases[10];   // массив указателей на функции

extern void reset( void * );

int main() {
   // ...
   reset( textCases[0] );  // ошибка: нет стандартного преобразования
                           // между int(*)() и void*
   return 0;
}
```

стандартного преобразования, с указателями на функции так поступать нельзя:

9.3.4. Ссылки

Фактический аргумент или формальный параметр функции могут быть ссылками. Как это влияет на правила преобразования типов?

Рассмотрим, что происходит, когда ссылкой является фактический аргумент. Его тип никогда не бывает ссылочным. Аргумент-ссылка трактуется как l-значение, тип которого

```
int i;
int& ri = i;
void print( int );

int main() {
   print( i );   // аргумент - это lvalue типа int
   print( ri );  // то же самое
   return 0;
}
```

совпадает с типом соответствующего объекта:

Фактический аргумент в обоих вызовах имеет тип int. Использование ссылки для его передачи во втором вызове не влияет на сам тип аргумента.

Стандартные преобразования и расширения типов, рассматриваемые компилятором, одинаковы для случаев, когда фактический аргумент является ссылкой на тип T и когда

```
int i;
int& ri = i;
void calc( double );

int main() {
   calc( i );   // стандартное преобразование между целым типом
                // и типом с плавающей точкой
   calc( ri );  // то же самое
   return 0;
}
```

он сам имеет такой тип. Например:

А как влияет на преобразования, применяемые к фактическому аргументу, формальный параметр-ссылка? Сопоставление дает следующие результаты:

- фактический аргумент подходит в качестве инициализатора параметра-ссылки.

```
void swap( int &, int & );

void manip( int i1, int i2 ) {
   // ...
   swap( i1, i2 );    // правильно: вызывается swap( int &, int & )
   // ...
   return 0;
}
```

В таком случае мы говорим, что между ними есть точное соответствие:

- фактический аргумент не может инициализировать параметр-ссылку. В такой ситуации точного соответствия нет, и аргумент нельзя использовать для вызова

```
int obj;
void frd( double & );
int main() {
   frd( obj );   // ошибка: параметр должен иметь иметь тип const double &
   return 0;
}
```

функции. Например:

Вызов функции `frd()` является ошибкой. Фактический аргумент имеет тип `int` и должен быть преобразован в тип `double`, чтобы соответствовать формальному параметру-ссылке. Результатом такой трансформации является временная переменная. Поскольку ссылка не имеет спецификатора `const`, то для ее инициализации такие переменные использовать нельзя.

Вот еще один пример, в котором между формальным параметром-ссылкой и

```
class B;
void takeB( B& );
B giveB();

int main() {
   takeB( giveB() );   // ошибка: параметр должен быть типа const B &
   return 0;
}
```

фактическим аргументом нет соответствия:

Вызов функции `takeB()` – ошибка. Фактический аргумент – это возвращаемое значение, т.е. временная переменная, которая не может быть использована для инициализации ссылки без спецификатора `const`.

В обоих случаях мы видим, что если формальный параметр-ссылка имеет спецификатор `const`, то между ним и фактическим аргументом может быть установлено точное соответствие.

Следует отметить, что и преобразование l-значения в r-значение, и инициализация ссылки считаются точными соответствиями. В данном примере первый вызов функции

```
void print( int );
void print( int& );

int iobj;
int &ri = iobj;

int main() {
   print( iobj );   // ошибка: неоднозначность
   print( ri );     // ошибка: неоднозначность
   print( 86 );     // правильно: вызывается print( int )
   return 0;
}
```

приводит к ошибке:

```
}
```

Объект `iobj` – это аргумент, для которого может быть установлено соответствие с обеими функциями `print()`, то есть вызов неоднозначен. То же относится и к следующей строке, где ссылка `ri` обозначает объект, соответствующий обеим функциям `print()`. С третьим вызовом, однако, все в порядке. Для него `print(int&)` не является устоявшей. Целая константа – это r-значение, так что она не может инициализировать параметр-ссылку. Единственной устоявшей функцией для вызова `print(86)` является `print(int)`, поэтому она и выбирается при разрешении перегрузки.

Короче говоря, если формальный параметр представляет собой ссылку, то для фактического аргумента точное соответствие устанавливается, если он может инициализировать ссылку, и не устанавливается в противном случае.

Упражнение 9.6

Назовите два тривиальных преобразования, допустимых при установлении точного соответствия.

Упражнение 9.7

```
(a) void print( int *, int );
    int arr[6];
    print( arr, 6 );   // вызов функции

(b) void manip( int, int );
    manip( 'a', 'z' ); // вызов функции

(c) int calc( int, int );
    double dobj;
    double = calc( 55.4, dobj ) // вызов функции

(d) void set( const int * );
    int *pi;
```

Каков ранг каждого из преобразований аргументов в следующих вызовах функций:

```
    set( pi ); // вызов функции
```

Упражнение 9.8

Какие из данных вызовов ошибочны из-за того, что не существует преобразования между типом фактического аргумента и формального параметра:

```
(a) enum Stat { Fail, Pass };
    void test( Stat );
    text( 0 ); // вызов функции

(b) void reset( void *);
    reset( 0 ); // вызов функции

(c) void set( void * );
    int *pi;
    set( pi ); // вызов функции

(d) #include <list>
    list<int> oper();
    void print( oper() ); // вызов функции

(e) void print( const int );
    int iobj;

    print( iobj ); // вызов функции
```

9.4. Детали разрешения перегрузки функций

В разделе 9.2 мы уже упоминали, что процесс разрешения перегрузки функций состоит из трех шагов:

1. Установить множество функций-кандидатов для разрешения данного вызова, а также свойства списка фактических аргументов.
2. Отобрать из множества кандидатов устоявшие функции – те, которые могут быть вызваны с данным списком фактических аргументов при учете их числа и типов.
3. Выбрать функцию, лучше всего соответствующую вызову, подвергнув ранжированию преобразования, которые необходимо применить к фактическим аргументам, чтобы привести их в соответствие с формальными параметрами устоявшей функции.

Теперь мы готовы к тому, чтобы изучить эти шаги более детально.

9.4.1. Функции-кандидаты

Функцией-кандидатом называется функция, имеющая то же имя, что и вызванная. Кандидаты отыскиваются двумя способами:

```
void f();
void f( int );
void f( double, double = 3.4 );
void f( char*, char* );

int main() {
   f( 5.6 );   // для разрешения этого вызова есть четыре кандидата
   return 0;
}
```

- объявление функции видимо в точке вызова. В следующем примере

все четыре функции `f()` удовлетворяют этому условию. Поэтому множество кандидатов содержит четыре элемента;

- если тип фактического аргумента объявлен внутри некоторого пространства имен, то функции-члены этого пространства, имеющие то же имя, что и вызванная

```
namespace NS {
   class C { /* ... */ };
   void takeC( C& );
}

// тип cobj - это класс C, объявленный в пространстве имен NS
NS::C obj;

int main() {
                  // в точке вызова не видна ни одна из функций takeC()
   takeC( cobj); // правильно: вызывается NS::takeC( C& ),
                  // потому что аргумент имеет тип NS::C, следовательно,
                  // принимается во внимание функция takeC(),
                  // объявленная в пространстве имен NS
   return 0;
}
```

функция, добавляются в множество кандидатов:

Таким образом, совокупность кандидатов является объединением множества функций, видимых в точке вызова, и множества функций, объявленных в том же пространстве имен, к которому принадлежат типы фактических аргументов.

При идентификации множества перегруженных функций, видимых в точке вызова, применимы уже рассмотренные ранее правила.

Функция, объявленная во вложенной области видимости, скрывает, а не перегружает одноименную функцию во внешней области. В такой ситуации кандидатами будут только функции из во вложенной области, т.е. такие, которые не скрыты при вызове. В следующем примере функциями-кандидатами, видимыми в точке вызова, являются

```
char* format( int );
void g() {
   char *format( double );
   char* format( char* );

   format(3);   // вызывается format( double )
}
```

`format(double)` и `format(char*)`:

Так как `format(int)`, объявленная в глобальной области видимости, скрыта, она не включается в множество функций-кандидатов.

Кандидаты могут быть введены с помощью using-объявлений, видимых в точке вызова:

```
namespace libs_R_us {
   int max( int, int );
   double max( double, double );
}

char max( char, char );

void func()
{
   // функции из пространства имен невидимы
   // все три вызова разрешаются в пользу глобальной функции max( char,
         char )
   max( 87, 65 );
   max( 35.5, 76.6 );
   max( 'J', 'L' );
}
```

Функции `max()`, определенные в пространстве имен `libs_R_us`, невидимы в точке вызова. Единственной видимой является функция `max()` из глобальной области; только она входит в множество функций-кандидатов и вызывается при каждом из трех обращений к `func()`. Мы можем воспользоваться using-объявлением, чтобы сделать видимыми функции `max()` из пространства имен `libs_R_us`. Куда поместить using-

```
char max( char, char );
```

объявление? Если включить его в глобальную область видимости:

```
using libs_R_us::max;   // using-объявление
```

то функции `max()` из `libs_R_us` добавляются в множество перегруженных функций, которое уже содержит `max()`, объявленную в глобальной области. Теперь все три функции видны внутри `func()` и становятся кандидатами. В этой ситуации вызовы

```
void func()
{
   max( 87, 65 );     // вызывается libs_R_us::max( int, int )
   max( 35.5, 76.6 ); // вызывается libs_R_us::max( double, double )
   max( 'J', 'L' );   // вызывается ::max( char, char )
}
```

`func()` разрешаются следующим образом:

Но что будет, если мы введем using-объявление в локальную область видимости функции

```
void func()
{
   // using-объявление
   using libs_R_us::max;

   // те же вызовы функций, что и выше
}
```

`func()`, как показано в данном примере?

Какие из функций max() будут включены в множество кандидатов? Напомним, что using-объявления вкладываются друг в друга. При наличии такого объявления в локальной области глобальная функция max(char, char) оказывается скрытой, так что в точке

```
libs_R_us::max( int, int );
```

вызова видны только

```
libs_R_us::max( double, double );
```

```
void func()
{
   // using-объявление
   // глобальная функция max( char, char ) скрыта
   using libs_R_us::max;

   max( 87, 65 );     // вызывается libs_R_us::max( int, int )
   max( 35.5, 76.6 ); // вызывается libs_R_us::max( double, double )
   max( 'J', 'L' );   // вызывается libs_R_us::max( int, int )
}
```

Они и являются кандидатами. Теперь вызовы func() разрешаются следующим образом:

Using-директивы также оказывают влияние на состав множества функций-кандидатов. Предположим, мы решили их использовать, чтобы сделать функции max() из пространства имен libs_R_us видимыми в func(). Если разместить следующую using-директиву в глобальной области видимости, то множество функций-кандидатов будет состоять из глобальной функции max(char, char) и функций max(int, int) и

```
namespace libs_R_us {
   int max( int, int );
   double max( double, double );
}

char max( char, char );
using namespace libs_R_us;  // using-директива

void func()
{
   max( 87, 65 );     // вызывается libs_R_us::max( int, int )
   max( 35.5, 76.6 ); // вызывается libs_R_us::max( double, double )
   max( 'J', 'L' );   // вызывается ::max( int, int )
}
```

max(double, double), объявленных в libs_R_us:

Что будет, если поместить using-директиву в локальную область видимости, как в следующем примере?

```
void func()
{
   // using-директива
   using namespace libs_R_us;

   // те же вызовы функций, что и выше
}
```

Какие из функций `max()` окажутся среди кандидатов? Напомним, что using-директива делает члены пространства имен видимыми, словно они были объявлены вне этого пространства, в той точке, где такая директива помещается. В нашем примере члены `libs_R_us` видимы в локальной области функции `func()`, как будто они объявлены вне пространства – в глобальной области. Отсюда следует, что множество перегруженных

```
max( char, char );
libs_R_us::max( int, int );
```

функций, видимых внутри `func()`, то же, что и раньше, т.е. включает в себя

```
libs_R_us::max( double, double );
```

В локальной или глобальной области видимости появляется using-директива, на

```
void func()
{
   using namespace libs_R_us;

   max( 87, 65 );     // вызывается libs_R_us::max( int, int )
   max( 35.5, 76.6 ); // вызывается libs_R_us::max( double, double )
   max( 'J', 'L' );   // вызывается ::max( int, int )
```

разрешение вызовов функции `func()` не влияет:

```
}
```

Итак, множество кандидатов состоит из функций, видимых в точке вызова, включая и те, которые введены using-объявлениями и using-директивами, а также из функций, объявленных в пространствах имен, ассоциированных с типами фактических аргументов. Например:

```
namespace basicLib {
   int print( int );
   double print( double );
}
namespace matrixLib {
   class matrix { /* ... */ };
   void print( const maxtrix & );
}
void display()
{
   using basicLib::print;

   matrixLib::matrix mObj;
   print( mObj );   // вызывается maxtrixLib::print( const maxtrix & )

   print( 87 );     // вызывается basicLib::print( const maxtrix & )
}
```

Кандидатами для `print(mObj)` являются введенные using-объявлением внутри `display()` функции `basicLib::print(int)` и `basicLib::print(double)`, поскольку они видимы в точке вызова. Так как фактический аргумент функции имеет тип `matrixLib::matrix`, то функция `print()`, объявленная в пространстве имен `matrixLib`, также будет кандидатом. Каковы функции-кандидаты для `print(87)`? Только `basicLib::print(int)` и `basicLib::print(double)`, видимые в точке вызова. Поскольку аргумент имеет тип `int`, дополнительное пространство имен в поисках других кандидатов не рассматривается.

9.4.2. Устоявшие функции

Устоявшая функция относится к числу кандидатов. В списке ее формальных параметров либо то же самое число элементов, что и в списке фактических аргументов вызванной функции, либо больше. В последнем случае для дополнительных параметров задаются значения по умолчанию, иначе функцию нельзя будет вызвать с данным числом аргументов. Чтобы функция считалась устоявшей, должно существовать преобразование каждого фактического аргумента в тип соответствующего формального параметра. (Такие преобразования были рассмотрены в разделе 9.3.)

В следующем примере для вызова `f(5.6)` есть две устоявшие функции: `f(int)` и

```
void f();
void f( int );
void f( double );
void f( char*, char* );

int main() {
   f( 5.6 );   // 2 устоявшие функции: f( int ) и f( double )
   return 0;
}
```

`f(double)`.

Функция `f(int)` устояла, так как она имеет всего один формальный параметр, что соответствует числу фактических аргументов в вызове. Кроме того, существует стандартное преобразование аргумента типа `double` в `int`. Функция `f(double)` также

устояла; она тоже имеет один параметр типа double, и он точно соответствует фактическому аргументу. Функции-кандидаты f() и f(char*, char*) исключены из списка устоявших, так как они не могут быть вызваны с одним аргументом.

В следующем примере единственной устоявшей функцией для вызова format(3) является format(double). Хотя кандидата format(char*) можно вызывать с одним аргументом, не существует преобразования из типа фактического аргумента int в тип

```
char* format( int );
void g() {
   // глобальная функция format( int ) скрыта
   char* format( double );
   char* format( char* );
   format(3);   // есть только одна устоявшая функция: format( double )
}
```

формального параметра char*, а следовательно, функция не может считаться устоявшей.

В следующем примере все три функции-кандидата оказываются устоявшими для вызова max() внутри func(). Все они могут быть вызваны с двумя аргументами. Поскольку фактические аргументы имеют тип int, они точно соответствуют формальным параметрам функции libs_R_us::max(int, int) и могут быть приведены к типам параметров функции libs_R_us::max(double, double) с помощью трансформации целых в плавающие, а также к типам параметров функции libs_R_us::max(char,

```
namespace libs_R_us {
   int max( int, int );
   double max( double, double );
}

// using-объявление
using libs_R_us::max;

char max( char, char );
void func()
{
   // все три функции max() являются устоявшими
   max( 87, 65 );   // вызывается using libs_R_us::max( int, int )
}
```

char) посредством преобразования целых типов.

Обратите внимание, что функция-кандидат с несколькими параметрами исключается из числа устоявших, как только выясняется, что один из фактических аргументов не может быть приведен к типу соответствующего формального параметра, пусть даже для всех остальных аргументов такое преобразование существует. В следующем примере функция min(char *, int) исключается из множества устоявших, поскольку нет возможности трансформации типа первого аргумента int в тип соответствующего параметра char *. И это происходит несмотря на то, что второй аргумент точно соответствует второму параметру.

```
extern double min( double, double );
extern double min( char*, int );

void func()
{
   // одна функция-кандидат min( double, double )
   min( 87, 65 );   // вызывается min( double, double )
}
```

Если после исключения из множества кандидатов всех функций с несоответствующим числом параметров и тех, для параметров которых не оказалось подходящего преобразования, не осталось устоявших, то обработка вызова функции заканчивается

```
void print( unsigned int );
void print( char* );
void print( char );

int *ip;
class SmallInt { /* ... */ };
SmallInt si;

int main() {
   print( ip );   // ошибка: нет устоявших функций: соответствие не
          найдено
   print( si );   // ошибка: нет устоявших функций: соответствие не
          найдено
   return 0;
}
```

ошибкой компиляции. В таком случае говорят, что соответствия не найдено.

9.4.3. Наилучшая из устоявших функция

Наилучшей считается та из устоявших функций, формальные параметры которой наиболее точно соответствуют типам фактических аргументов. Для любой такой функции преобразования типов, применяемые к каждому аргументу, ранжируются для определения степени его соответствия параметру. (В разделе 6.2 описаны поддерживаемые преобразования типов.) Наилучшей из устоявших называют функцию, для которой одновременно выполняются два условия:

- преобразования, примененные к аргументам, *не хуже* преобразований, необходимых для вызова любой другой устоявшей функции;
- хотя бы для одного аргумента примененное преобразование *лучше*, чем для того же аргумента в любой другой устоявшей функции.

Может оказаться так, что для приведения фактического аргумента к типу соответствующего формального параметра нужно выполнить несколько преобразований. Так, в следующем примере

```
int arr[3];
void putValues(const int *);

int main() {
   putValues(arr);   // необходимо 2 преобразования
                     // массив в указатель + преобразование спецификатора
   return 0;
}
```

для приведения аргумента `arr` от типа “массив из трех `int`” к типу “указатель на `const int`” применяется последовательность преобразований:

1. Преобразование массива в указатель, которое трансформирует массив из трех `int` в указатель на `int`.
2. Преобразование спецификатора, которое трансформирует указатель на `int` в указатель на `const int`.

Поэтому было бы более правильно говорить, что для приведения фактического аргумента к типу формального параметра устоявшей функции требуется *последовательность преобразований*. Поскольку применяется не одна, а несколько трансформаций, то на третьем шаге процесса разрешения перегрузки функции на самом деле ранжируются последовательности преобразований.

Рангом такой последовательности считается ранг самой плохой из входящих в нее трансформаций. Как объяснялось в разделе 9.2, преобразования типов ранжируются следующим образом: точное соответствие лучше расширения типа, а расширение типа лучше стандартного преобразования. В предыдущем примере оба изменения имеют ранг точного соответствия. Поэтому и у всей последовательности такой же ранг.

Такая совокупность состоит из нескольких преобразований, применяемых в указанном порядке:

преобразование l-значения ->

расширение типа или стандартное преобразование ->

преобразование спецификаторов

Термин *преобразование l-значения* относится к первым трем трансформациям из категории точных соответствий, рассмотренных в разделе 9.2: преобразование l-значения в r-значение, преобразование массива в указатель и преобразование функции в указатель. Последовательность трансформаций состоит из нуля или одного преобразования l-значения, за которым следует нуль или одно расширение типа или стандартное преобразование, и наконец нуль или одно преобразование спецификаторов. Для приведения фактического аргумента к типу формального параметра может быть применено только одна трансформация каждого вида.

Описанная последовательность называется последовательностью *стандартных* преобразований. Существует также последовательность *определенных пользователем* преобразований, которая связана с функцией-конвертером, являющейся членом класса. (Конвертеры и последовательности определенных пользователем преобразований рассматриваются в главе 15.)

Каковы последовательности изменений фактических аргументов в следующем примере?

```
namespace libs_R_us {
   int max( int, int );
   double max( double, double );
}

// using-объявление
using libs_R_us::max;

void func()
{
   char c1, c2;
   max( c1, c2 );   // вызывается libs_R_us::max( int, int )
}
```

Аргументы в вызове функции `max()` имеют тип `char`. Последовательность преобразований аргументов при вызове функции `libs_R_us::max(int,int)` следующая:

1a. Так как аргументы передаются по значению, то с помощью преобразования l-значения в r-значение извлекаются значения аргументов `c1` и `c2`.

2a. С помощью расширения типа аргументы трансформируются из `char` в `int`.

Последовательность преобразований аргументов при вызове функции `libs_R_us::max(double,double)` следующая:

1b. С помощью преобразования l-значения в r-значение извлекаются значения аргументов `c1` и `c2`.

2b. Стандартное преобразование между целым и плавающим типом приводит аргументы от типа `char` к типу `double`.

Ранг первой последовательности – расширение типа (самое худшее из примененных изменений), тогда как ранг второй – стандартное преобразование. Так как расширение типа лучше, чем преобразование, то в качестве наилучшей из устоявших для данного вызова выбирается функция `libs_R_us::max(int,int)`.

Если ранжирование последовательностей преобразований аргументов не может выявить единственной устоявшей функции, то вызов считается неоднозначным. В данном примере для обоих вызовов `calc()` требуется такая последовательность:

1. Преобразование l-значения в r-значение для извлечения значений аргументов `i` и `j`.
2. Стандартное преобразование для приведения типов фактических аргументов к типам соответствующих формальных параметров.

Поскольку нельзя сказать, какая из этих последовательностей лучше другой, вызов неоднозначен:

```
int i, j;
extern long calc( long, long );
extern double calc( double, double );

void jj() {

   // ошибка: неоднозначность, нет наилучшего соответствия
   calc( i, j );

}
```

Преобразование спецификаторов (добавление спецификатора `const` или `volatile` к типу, который адресует указатель) имеет ранг точного соответствия. Однако, если две последовательности трансформаций отличаются только тем, что в конце одной из них есть дополнительное преобразование спецификаторов, то последовательность без него

```
void reset( int * );
void reset( const int * );

int* pi;

int main() {
   reset( pi );   // без преобразования спецификаторов лучше:
                  // выбирается reset( int * )
   return 0;
}
```

считается лучше. Например:

Последовательность стандартных преобразований, примененная к фактическому аргументу для первой функции-кандидата `reset(int*)`, – это точное соответствие, требуется лишь переход от l-значения к r-значению, чтобы извлечь значение аргумента. Для второй функции-кандидата `reset(const int *)` также применяется трансформация l-значения в r-значение, но за ней следует еще и преобразование спецификаторов для приведения результирующего значения от типа “указатель на `int`” к типу “указатель на `const int`”. Обе последовательности представляют собой точное соответствие, но неоднозначности при этом не возникает. Так как вторая последовательность отличается от первой наличием трансформации спецификаторов в конце, то последовательность без такого преобразования считается лучшей. Поэтому наилучшей из устоявших функций будет `reset(int*)`.

Вот еще пример, в котором приведение спецификаторов влияет на то, какая

```
int extract( void * );
int extract( const void * );

int* pi;

int main() {
   extract( pi );  // выбирается extract( void * )
   return 0;
}
```

последовательность будет выбрана:

Здесь для вызова есть две устоявших функции: `extract(void*)` и `extract(const void*)`. Последовательность преобразований для функции `extract(void*)` состоит из трансформации l-значения в r-значение для извлечения значения аргумента, сопровождаемого стандартным преобразованием указателя: из указателя на `int` в указатель на `void`. Для функции `extract(const void*)` такая последовательность отличается от первой дополнительным преобразованием спецификаторов для приведения типа результата от указателя на `void` к указателю на `const void`. Поскольку последовательности различаются лишь этой трансформацией, то первая выбирается как более подходящая и, следовательно, наилучшей из устоявших будет функция `extract(const void*)`.

Спецификаторы `const` и `volatile` влияют также на ранжирование инициализации параметров-ссылок. Если две такие инициализации отличаются только добавлением спецификатора `const` и `volatile`, то инициализация без дополнительной спецификации

```
#include <vector>
void manip( vector<int> & );
void manip( const vector<int> & );

vector<int> f();
extern vector<int> vec;

int main() {
   manip( vec );   // выбирается manip( vector<int> & )
   manip( f() );   // выбирается manip( const vector<int> & )
   return 0;
}
```

считается лучшей при разрешении перегрузки:

В первом вызове инициализация ссылок для вызова любой функции является точным соответствием. Но этот вызов все же не будет неоднозначным. Так как обе инициализации одинаковы во всем, кроме наличия дополнительной спецификации `const` во втором случае, то инициализация без такой спецификации считается лучше, поэтому перегрузка будет разрешена в пользу устоявшей функции `manip(vector<int>&)`.

Для второго вызова существует только одна устоявшая функция `manip(const vector<int>&)`. Поскольку фактический аргумент является временной переменной, содержащей результат, возвращенный `f()`, то такой аргумент представляет собой r-значение, которое нельзя использовать для инициализации неконстантного формального параметра-ссылки функции `manip(vector<int>&)`. Поэтому наилучшей является единственная устоявшая `manip(const vector<int>&)`.

Разумеется, у функций может быть несколько фактических аргументов. Выбор наилучшей из устоявших должен производиться с учетом ранжирования

```
extern int ff( char*, int );
extern int ff( int, int );

int main() {

   ff( 0, 'a' );  // ff( int, int )
   return 0;
```

последовательностей преобразований всех аргументов. Рассмотрим пример:

```
}
```

Функция `ff()`, принимающая два аргумента типа `int`, выбирается в качестве наилучшей из устоявших по следующим причинам:

1. ее первый аргумент лучше. `0` дает точное соответствие с формальным параметром типа `int`, тогда как для установления соответствия с параметром типа `char *` требуется стандартное преобразование указателя;
2. ее второй аргумент имеет тот же ранг. К аргументу `'a'` типа `char` для установления соответствия со вторым формальным параметром любой из двух функций должна быть применена последовательность преобразований, имеющая ранг расширения типа.

```
int compute( const int&, short );
int compute( int&, double );

extern int iobj;
int main() {
   compute( iobj, 'c' );  // compute( int&, double )
   return 0;
```

Вот еще один пример:

```
}
```

Обе функции `compute( const int&, short )` и `compute( int&, double )` устояли. Вторая выбирается в качестве наилучшей по следующим причинам:

1. ее первый аргумент лучше. Инициализация ссылки для первой устоявшей функции хуже потому, что она требует добавления спецификатора `const`, не нужного для второй функции;
2. ее второй аргумент имеет тот же ранг. К аргументу `'c'` типа `char` для установления соответствия со вторым формальным параметром любой из двух функций должна быть применена последовательность трансформаций, имеющая ранг стандартного преобразования.

9.4.4. Аргументы со значениями по умолчанию

Наличие аргументов со значениями по умолчанию способно расширить множество устоявших функций. Устоявшими являются функции, которые вызываются с данным списком фактических аргументов. Но такая функция может иметь больше формальных параметров, чем задано фактических аргументов, в том случае, когда для каждого неуказанного параметра есть некое значение по умолчанию:

```
extern void ff( int );
extern void ff( long, int = 0 );

int main() {
   ff( 2L );   // соответствует ff( long, 0 );

   ff( 0, 0 ); // соответствует ff( long, int );
   ff( 0 );    // соответствует ff( int );
   ff( 3.14 ); // ошибка: неоднозначность

}
```

Для первого и третьего вызовов функция `ff()` является устоявшей, хотя передан всего один фактический аргумент. Это обусловлено следующими причинами:

1. для второго формального параметра есть значение по умолчанию;
2. первый параметр типа `long` точно соответствует фактическому аргументу в первом вызове и может быть приведен к типу аргумента в третьем вызове за счет последовательности, имеющей ранг стандартного преобразования.

Последний вызов является неоднозначным, поскольку обе устоявших функции могут быть выбраны, если применить стандартное преобразование к первому аргументу. Функции `ff(int)` не отдается предпочтение только потому, что у нее один параметр.

Упражнение 9.9

Объясните, что происходит при разрешении перегрузки для вызова функции `compute()` внутри `main()`. Какие функции являются кандидатами? Какие из них устоят после первого шага? Какие последовательности преобразований надо применить к фактическому аргументу, чтобы он соответствовал формальному параметру для каждой

```
namespace primerLib {
   void compute();
   void compute( const void * );
}

using primerLib::compute;
void compute( int );
void compute( double, double = 3.4 );
void compute( char*, char* = 0 );

int main() {
   compute( 0 );
   return 0;
}
```

устоявшей функции? Какая функция будет наилучшей из устоявших?

Что будет, если using-объявление поместить внутрь `main()` перед вызовом `compute()`? Ответьте на те же вопросы.

10

10. Шаблоны функций

В этой главе рассказывается, что такое шаблон функции, как его определять и использовать. Это довольно просто, и многие программисты применяют шаблоны, определенные в стандартной библиотеке, даже не понимая, с чем они работают. Только пользователи, хорошо знающие язык C++, самостоятельно определяют и применяют шаблоны функций так, как здесь описано. Поэтому материал данной главы следует рассматривать как переход к более сложным аспектам C++. Мы начнем с рассказа о том, что такое шаблон функции и как его определять, затем на простом примере проиллюстрируем использование шаблонов. Далее мы перейдем к темам, требующим больших знаний. Сначала посмотрим на усложненные примеры применения шаблонов, затем подробно остановимся на выведении (deduction) их аргументов и покажем, как их можно задавать при конкретизации (instantiation) шаблона функции. После этого мы посмотрим, каким образом компилятор конкретизирует шаблоны и какие требования предъявляются в этой связи к организации наших программ, а также обсудим, как определить специализацию для такой конкретизации. Затем в данной главе будут изложены вопросы, представляющие интерес для проектировщиков шаблонов функций. Мы объясним, как можно перегружать шаблоны и как применительно к ним работает разрешение перегрузки. Мы также расскажем о разрешении имен в определениях шаблонов функций и покажем, как можно определять шаблоны в пространствах имен. Глава завершается развернутым примером.

10.1. Определение шаблона функции

Иногда может показаться, что сильно типизированный язык создает препятствия для реализации совсем простых функций. Например, хотя следующий алгоритм функции `min()` тривиален, сильная типизация требует, чтобы его разновидности были

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

double min( double a, double b ) {
   return a < b ? a : b;
```

реализованы для всех типов, которые мы собираемся сравнивать:

```
}
```

Заманчивую альтернативу явному определению каждого экземпляра функции `min()` представляет использование макросов, расширяемых препроцессором:

```
#define min(a, b) ((a) < (b) ? (a) : (b))
```

Но этот подход таит в себе потенциальную опасность. Определенный выше макрос правильно работает при простых обращениях к `min()`, например:

```
min( 10, 20 );
min( 10.0, 20.0 );
```

но может преподнести сюрпризы в более сложных случаях: такой механизм ведет себя не как вызов функции, он лишь выполняет текстовую подстановку аргументов. В результате значения обоих аргументов оцениваются *дважды*: один раз при сравнении a и b, а

```
#include <iostream>

#define min(a,b) ((a) < (b) ? (a) : (b))

const int size = 10;
int ia[size];

int main() {
   int elem_cnt = 0;
   int *p = &ia[0];

   // подсчитать число элементов массива
   while ( min(p++,&ia[size]) != &ia[size] )
      ++elem_cnt;

   cout << "elem_cnt : "   << elem_cnt
        << "\texpecting: " << size << endl;
   return 0;
}
```

второй – при вычислении возвращаемого макросом результата:

На первый взгляд, эта программа подсчитывает количество элементов в массиве ia целых чисел. Но в этом случае макрос min() расширяется неверно, поскольку операция постинкремента применяется к аргументу-указателю дважды при каждой подстановке. В результате программа печатает строку, свидетельствующую о неправильных вычислениях:

```
elem_cnt : 5    expecting: 10
```

Шаблоны функций предоставляют в наше распоряжение механизм, с помощью которого можно сохранить семантику определений и вызовов функций (инкапсуляция фрагмента кода в одном месте программы и гарантированно однократное вычисление аргументов), не принося в жертву сильную типизацию языка C++, как в случае применения макросов.

Шаблон дает алгоритм, используемый для автоматической генерации экземпляров функций с различными типами. Программист *параметризует* все или только некоторые типы в интерфейсе функции (т.е. типы формальных параметров и возвращаемого значения), оставляя ее тело неизменным. Функция хорошо подходит на роль шаблона, если ее реализация остается инвариантной на некотором множестве экземпляров, различающихся типами данных, как, скажем, в случае min().

Так определяется шаблон функции min():

```
template <class Type>
   Type min2( Type a, Type b ) {
      return a < b ? a : b;
}

int main() {
   // правильно: min( int, int );
   min( 10, 20 );

   // правильно: min( double, double );
   min( 10.0, 20.0 );
   return 0;
}
```

Если вместо макроса препроцессора `min()` подставить в текст предыдущей программы этот шаблон, то результат будет правильным:

```
elem_cnt : 10    expecting: 10
```

(В стандартной библиотеке C++ есть шаблоны функций для многих часто используемых алгоритмов, например для `min()`. Эти алгоритмы описываются в главе 12. А в данной вводной главе мы приводим собственные упрощенные версии некоторых алгоритмов из стандартной библиотеки.)

Как объявление, так и определение шаблона функции всегда должны начинаться с ключевого слова `template`, за которым следует список разделенных запятыми идентификаторов, заключенный в угловые скобки '<' и '>', – *список параметров шаблона*, обязательно непустой. У шаблона могут быть *параметры-типы*, представляющие некоторый тип, и *параметры-константы*, представляющие фиксированное константное выражение.

Параметр-тип состоит из ключевого слова `class` или ключевого слова `typename`, за которым следует идентификатор. Эти слова всегда обозначают, что последующее имя относится к встроенному или определенному пользователем типу. Имя параметра шаблона выбирает программист. В приведенном примере мы использовали имя `Type`, но

```
template <class Glorp>
   Glorp min2( Glorp a, Glorp b ) {
      return a < b ? a : b;
}
```

могли выбрать и любое другое:

При конкретизации (порождении конкретного экземпляра) шаблона вместо параметра-типа подставляется фактический встроенный или определенный пользователем тип. Любой из типов `int`, `double`, `char*`, `vector<int>` или `list<double>` является допустимым аргументом шаблона.

Параметр-константа выглядит как обычное объявление. Он говорит о том, что вместо имени параметра должно быть подставлено значение константы из определения шаблона. Например, `size` – это параметр-константа, который представляет размер массива `arr`:

```
template <class Type, int size>
   Type min( Type (&arr) [size] );
```

Вслед за списком параметров шаблона идет объявление или определение функции. Если не обращать внимания на присутствие параметров в виде спецификаторов типа или констант, то определение шаблона функции выглядит точно так же, как и для обычных

```
template <class Type, int size>
Type min( const Type (&r_array)[size] )
{
   /* параметризованная функция для отыскания
    * минимального значения в массиве */
   Type min_val = r_array[0];
   for ( int i = 1; i < size; ++i )
      if ( r_array[i] < min_val )
         min_val = r_array[i];

   return min_val;
}
```

функций:

В этом примере `Type` определяет тип значения, возвращаемого функцией `min()`, тип параметра `r_array` и тип локальной переменной `min_val`; `size` задает размер массива `r_array`. В ходе работы программы при использовании функции `min()` вместо `Type` могут быть подставлены любые встроенные и определенные пользователем типы, а вместо `size` – те или иные константные выражения. (Напомним, что работать с функцией можно двояко: вызвать ее или взять ее адрес).

Процесс подстановки типов и значений вместо параметров называется *конкретизацией шаблона*. (Подробнее мы остановимся на этом в следующем разделе.)

Список параметров нашей функции `min()` может показаться чересчур коротким. Как было сказано в разделе 7.3, когда параметром является массив, передается указатель на его первый элемент, первая же размерность фактического аргумента-массива внутри определения функции неизвестна. Чтобы обойти эту трудность, мы объявили первый параметр `min()` как ссылку на массив, а второй – как его размер. Недостаток подобного подхода в том, что при использовании шаблона с массивами одного и того же типа `int`, но разных размеров генерируются (или конкретизируются) различные экземпляры функции `min()`.

Имя параметра разрешено употреблять внутри объявления или определения шаблона. Параметр-тип служит спецификатором типа; его можно использовать точно так же, как спецификатор любого встроенного или пользовательского типа, например в объявлении переменных или в операциях приведения типов. Параметр-константа применяется как константное значение – там, где требуются константные выражения, например для задания размера в объявлении массива или в качестве начального значения элемента перечисления.

```
// size определяет размер параметра-массива и инициализирует
// переменную типа const int
template <class Type, int size>
   Type min( const Type (&r_array)[size] )
{
   const int loc_size = size;
   Type loc_array[loc_size];
   // ...
}
```

Если в глобальной области видимости объявлен объект, функция или тип с тем же именем, что у параметра шаблона, то глобальное имя оказывается скрытым. В следующем примере тип переменной tmp не double, а тот, что у параметра шаблона

```
typedef double Type;
template <class Type>
   Type min( Type a, Type b )
{
   // tmp имеет тот же тип, что параметр шаблона Type, а не заданный
   // глобальным typedef
   Type tm = a < b ? a : b;
   return tmp;
}
```

Type:

Объект или тип, объявленные внутри определения шаблона функции, не могут иметь то

```
template <class Type>
   Type min( Type a, Type b )
{
   // ошибка: повторное объявление имени Type, совпадающего с именем
   // параметра шаблона
   typedef double Type;
   Type tmp = a < b ? a : b;
   return tmp;
}
```

же имя, что и какой-то из параметров:

Имя параметра-типа шаблона можно использовать для задания типа возвращаемого

```
// правильно: T1 представляет тип значения, возвращаемого min(),
// а T2 и T3 – параметры-типы этой функции
template <class T1, class T2, class T3>
```

значения:

```
   T1 min( T2, T3 );
```

В одном списке параметров некоторое имя разрешается употреблять только один раз. Например, следующее определение будет помечено как ошибка компиляции:

```
// ошибка: неправильное повторное использование имени параметра Type
template <class Type, class Type>

   Type min( Type, Type );
```

Однако одно и то же имя можно многократно применять внутри объявления или

```
// правильно: повторное использование имени Type внутри шаблона
template <class Type>
```

определения шаблона:

```
template <class Type>

   Type min( Type, Type );
   Type max( Type, Type );
```

Имена параметров в объявлении и определении не обязаны совпадать. Так, все три

```
// все три объявления min() относятся к одному и тому же шаблону функции

// опережающие объявления шаблона
template <class T> T min( T, T );
template <class U> U min( U, U );

// фактическое определение шаблона
template <class Type>
```

объявления `min()` относятся к одному и тому же шаблону функции:

```
   Type min( Type a, Type b ) { /* ... */ }
```

Количество появлений одного и того же параметра шаблона в списке параметров функции не ограничено. В следующем примере `Type` используется для представления

```
#include <vector>
// правильно: Type используется неоднократно в списке параметров шаблона
template <class Type>
```

двух разных параметров:

```
   Type sum( const vector<Type> &, Type );
```

Если шаблон функции имеет несколько параметров-типов, то каждому из них должно

```
// правильно: ключевые слова typename и class могут перемежаться
template <typename T, class U>
   T minus( T*, U );

// ошибка: должно быть <typename T, class U> или
//                     <typename T, typename U>
template <typename T, U>
```

предшествовать ключевое слово `class` или `typename`:

```
   T sum( T*, U );
```

В списке параметров шаблона функции ключевые слова typename и class имеют одинаковый смысл и, следовательно, взаимозаменяемы. Любое из них может использоваться для объявления разных параметров-типов шаблона в одном и том же списке (как было продемонстрировано на примере шаблона функции minus()). Для обозначения параметра-типа более естественно, на первый взгляд, употреблять ключевое слово typename, а не class, ведь оно ясно указывает, что за ним следует имя типа. Однако это слово было добавлено в язык лишь недавно, как часть стандарта C++, поэтому в старых программах вы скорее всего встретите слово class. (Не говоря уже о том, что class короче, чем typename, а человек по природе своей ленив.)

Ключевое слово typename упрощает разбор определений шаблонов. (Мы лишь кратко остановимся на том, зачем оно понадобилось. Желающим узнать об этом подробнее рекомендуем обратиться к книге Страуструпа “Design and Evolution of C++”.)

При таком разборе компилятор должен отличать выражения-типы от тех, которые таковыми не являются; выявить это не всегда возможно. Например, если компилятор встречает в определении шаблона выражение Parm::name и если Parm – это параметр-тип, представляющий класс, то следует ли считать, что name представляет член-тип

```
template <class Parm, class U>
   Parm minus( Parm* array, U value )
{
   Parm::name * p;   // это объявление указателя или умножение?
                     // На самом деле умножение
```

класса Parm?

```
}
```

Компилятор не знает, является ли name типом, поскольку определение класса, представленного параметром Parm, недоступно до момента конкретизации шаблона. Чтобы такое определение шаблона можно было разобрать, пользователь должен подсказать компилятору, какие выражения включают типы. Для этого служит ключевое слово typename. Например, если мы хотим, чтобы выражение Parm::name в шаблоне функции minus() было именем типа и, следовательно, вся строка трактовалась как

```
template <class Parm, class U>
   Parm minus( Parm* array, U value )
{
   typename Parm::name * p;   // теперь это объявление указателя
```

объявление указателя, то нужно модифицировать текст следующим образом:

```
}
```

Ключевое слово typename используется также в списке параметров шаблона для указания того, что параметр является типом.

Шаблон функции можно объявлять как inline или extern – как и обычную функцию. Спецификатор помещается после списка параметров, а не перед словом template.

```
// правильно: спецификатор после списка параметров
template <typename Type>
   inline
   Type min( Type, Type );

// ошибка: спецификатор inline не на месте
inline
template <typename Type>
   Type min( Array<Type>, int );
```

Упражнение 10.1

Определите, какие из данных определений шаблонов функций неправильны. Исправьте

```
(a) template <class T, U, class V>
       void foo( T, U, V );

(b) template <class  T>
       T foo( int *T );

(c) template <class T1, typename T2, class T3>
       T1 foo( T2, T3 );

(d) inline template <typename T>
       T foo( T, unsigned int* );

(e) template <class myT, class myT>
       void foo( myT, myT );

(f) template <class T>
       foo( T, T );

(g) typedef char Ctype;
    template <class Ctype>
       Ctype foo( Ctype a, Ctype b );
```

ошибки.

Упражнение 10.2

```
(a) template <class Type>
       Type bar( Type, Type );

    template <class Type>
       Type bar( Type, Type );

(b) template <class T1, class T2>
       void bar( T1, T2 );

    template <typename C1, typename C2>
       void bar( C1, C2 );
```

Какие из повторных объявлений шаблонов ошибочны? Почему?

Упражнение 10.3

Перепишите функцию putValues() из раздела 7.3.3 в виде шаблона. Параметризуйте его так, чтобы было два параметра шаблона (для типа элементов массива и для размера массива) и один параметр функции, являющийся ссылкой на массив. Напишите определение шаблона функции.

10.2. Конкретизация шаблона функции

Шаблон функции описывает, как следует строить конкретные функции, если задано множество фактических типов или значений. Процесс конструирования называется *конкретизацией шаблона*. Выполняется он неявно, как побочный эффект вызова или взятия адреса шаблона функции. Например, в следующей программе min() конкретизируется дважды: один раз для массива из пяти элементов типа int, а другой –

```
// определение шаблона функции min()
// с параметром-типом Type и параметром-константой size

template <typename Type, int size>
   Type min( Type (&r_array)[size] )
{

   Type min_val = r_array[0];
   for ( int i = 1; i < size; ++i )
      if ( r_array[i] < min_val )
         min_val = r_array[i];

   return min_val;
}

// size не задан -- ok
// size = число элементов в списке инициализации
int ia[] = { 10, 7, 14, 3, 25 };

double da[6] = { 10.2, 7.1, 14.5, 3.2, 25.0, 16.8 };

#include <iostream>
int main()
{
   // конкретизация min() для массива из 5 элементов типа int
   // подставляется Type => int, size => 5
   int i = min( ia );
   if ( i != 3 )
      cout << "??oops: integer min() failed\n";
   else cout << "!!ok: integer min() worked\n";

   // конкретизация min() для массива из 6 элементов типа double
   // подставляется Type => double, size => 6
   double d = min( da );
   if ( d != 3.2 )
      cout << "??oops: double min() failed\n";
   else cout << "!!ok: double min() worked\n";
   return 0;
```

для массива из шести элементов типа double:

```
}
```

Вызов

```
int i = min( ia );
```

приводит к конкретизации следующего экземпляра функции min(), в котором Type

```
int min( int (&r_array)[5] )
{
   int min_val = r_array[0];
   for ( int i = 1; i < 5; ++i )
      if ( r_array[i] < min_val )
         min_val = r_array[i];

   return min_val;
}
```

заменено на int, а size на 5:

Аналогично вызов

```
double d = min( da );
```

конкретизирует экземпляр min(), в котором Type заменено на double, а size на 6:

В качестве формальных параметров шаблона функции используются параметр-тип и параметр-константа. Для определения фактического типа и значения константы, которые надо подставить в шаблон, исследуются фактические аргументы, переданные при вызове функции. В нашем примере для идентификации аргументов шаблона при конкретизации используются тип ia (массив из пяти int) и da (массив из шести double). Процесс определения типов и значений аргументов шаблона по известным фактическим аргументам функции называется *выведением (deduction) аргументов шаблона*. (В следующем разделе мы расскажем об этом подробнее. А в разделе 10.4 речь пойдет о возможности явного задания аргументов.)

Шаблон конкретизируется либо при вызове, либо при взятии адреса функции. В следующем примере указатель pf инициализируется адресом конкретизированного экземпляра шаблона. Его аргументы определяются путем исследования типа параметра

```
template <typename Type, int size>
   Type min( Type (&p_array)[size] ) { /* ... */ }

// pf указывает на int min( int (&)[10] )
```

функции, на которую указывает pf:

```
int (*pf)(int (&)[10]) = &min;
```

Тип pf – это указатель на функцию с параметром типа int(&)[10], который определяет тип аргумента шаблона Type и значение аргумента шаблона size при конкретизации min(). Аргумент шаблона Type будет иметь тип int, а значением аргумента шаблона size будет 10. Конкретизированная функция представляется как min(int(&)[10]), и указатель pf адресует именно ее.

Когда берется адрес шаблона функции, контекст должен быть таким, чтобы можно было однозначно определить типы и значения аргументов шаблона. Если сделать это не удается, компилятор выдает сообщение об ошибке:

```
template <typename Type, int size>
   Type min( Type (&r_array)[size] ) { /* ... */ }

typedef int (&rai)[10];
typedef double (&rad)[20];

void func( int (*)(rai) );
void func( double (*)(rad) );

int main() {

// ошибка: как конкретизировать min()?
   func( &min );

}
```

Функция `func()` перегружена и тип ее параметра не позволяет однозначно определить ни аргумент шаблона `Type`, ни значение аргумента шаблона `size`. Результатом

```
min( int (*)(int(&)[10]) )
```

конкретизации вызова `func()` может быть любая из следующих функций:

```
min( double (*)(double(&)[20]) )
```

Поскольку однозначно определить аргументы функции `func()` нельзя, взятие адреса конкретизированного шаблона в таком контексте приводит к ошибке компиляции.

Этого можно избежать, если использовать явное приведение типов для указания типа

```
int main() {
   // правильно: с помощью явного приведения указывается тип аргумента
   func( static_cast< double(*)(rad) >(&min) );
}
```

аргумента:

Лучше, однако, применять явное задание аргументов шаблона, как будет показано в разделе 10.4.

10.3. Вывод аргументов шаблона A

При вызове шаблона функции типы и значения его аргументов определяются путем исследования типов фактических аргументов функции. Этот процесс называется *выводом аргументов шаблона*.

```
template <class Type, int size>
```

Параметром функции в шаблоне `min()` является ссылка на массив элементов типа `Type`:

```
   Type min( Type (&r_array)[size] ) { /* ... */ }
```

Для сопоставления с формальным параметром функции фактический аргумент также должен быть l-значением, представляющим тип массива. Следующий вызов ошибочен,

```
void f( int pval[9] ) {
   // ошибка: Type (&)[] != int*
   int jval = min( pval );
```

так как pval имеет тип int*, а не является l-значением типа “массив int”.

```
}
```

При выводе аргументов шаблона не принимается во внимание тип значения, возвращаемого конкретизированным шаблоном функции. Например, если вызов min()

```
double da[8] = { 10.3, 7.2, 14.0, 3.8, 25.7, 6.4, 5.5, 16.8 };
```

записан так:

```
int i1 = min( da );
```

то конкретизированный экземпляр min() имеет параметр типа “указатель на массив из восьми double” и возвращает значение типа double. Перед инициализацией i1 это значение приводится к типу int. Однако тот факт, что результат вызова min() используется для инициализации объекта типа int, не влияет на вывод аргументов шаблона.

Чтобы процесс такого вывода завершился успешно, тип фактического аргумента функции не обязательно должен совпадать с типом соответствующего формального параметра. Допустимы три вида преобразований типа: трансформация l-значения, преобразование спецификаторов и приведение к базовому классу, конкретизированному из шаблона класса. Рассмотрим последовательно каждое из них.

Напомним, что трансформация l-значения – это либо преобразование l-значения в r-значение, либо преобразование массива в указатель, либо преобразование функции в указатель (все они рассматривались в разделе 9.3). Для иллюстрации влияния такой трансформации на вывод аргументов шаблона рассмотрим функцию min2() с одним параметром шаблона Type и двумя параметрами функции. Первый параметр min2() – это указатель на тип Type*. size теперь не является параметром шаблона, как в определении min(), вместо этого он стал параметром функции, а его значение должно

```
template <class Type>
   // первый параметр имеет тип Type*
   Type min2( Type* array, int size )
{
   Type min_val = array[0];
   for ( int i = 1; i < size; ++i )
      if ( array[i] < min_val )
         min_val = array[i];

   return min_val;
```

быть явно передано при вызове:

```
}
```

min2() можно вызвать, передав в качестве первого аргумента массив из четырех int, как

```
int ai[4] = { 12, 8, 73, 45 };

int main() {
   int size = sizeof (ai) / sizeof (ai[0]);

   // правильно: преобразование массива в указатель
   min2( ai, size );
}
```

в следующем примере:

Фактический аргумент функции ai имеет тип “массив из четырех int” и не совпадает с типом соответствующего формального параметра Type*. Однако, поскольку преобразование массива в указатель допустимо, то аргумент ai приводится к типу int* еще до вывода аргумента шаблона Type, для которого затем выводится тип int, и шаблон конкретизирует функцию min2(int*, int).

Преобразование спецификаторов добавляет const или volatile к указателям (такие трансформации также рассматривались в разделе 9.3). Для иллюстрации влияния преобразования спецификаторов на вывод аргументов шаблона рассмотрим min3() с

```
template <class Type>
   // первый параметр имеет тип const Type*
   Type min3( const Type* array, int size ) {
   // ...
}
```

первым параметром функции типа const Type*:

min3() можно вызвать, передав int* в качестве первого фактического аргумента, как в

```
int *pi = &ai;
// правильно: приведение спецификаторов к типу const int*
```

следующем примере:

```
int i = min3( pi, 4 );
```

Фактический аргумент функции pi имеет тип “указатель на int” и не совпадает с типом формального параметра const Type*. Однако, поскольку преобразование спецификаторов допустимо, то он приводится к типу const int* еще до вывода аргумента шаблона Type, для которого затем выводится тип int, и шаблон конкретизирует функцию min3(const int*, int).

Теперь обратимся к преобразованию в базовый класс, конкретизированный из шаблона класса. Вывод аргументов шаблона можно выполнить, если тип формального параметра функции является таким шаблоном, а фактический аргумент – базовый класс, конкретизированный из него. Чтобы проиллюстрировать такое преобразование, рассмотрим новый шаблон функции min4() с параметром типа Array<Type>&, где Array – это шаблон класса, определенный в разделе 2.5. (В главе 16 шаблоны классов обсуждаются во всех деталях.)

```
template <class Type>
   class Array { /* ... */ }

template <class Type>
   Type min4( Array<Type>& array )
{
   Type min_val = array[0];
   for ( int i = 1; i < array.size(); ++i )
      if ( array[i] < min_val )
         min_val = array[i];

   return min_val;
}
```

`min4()` можно вызвать, передав в качестве первого аргумента `ArrayRC<int>`, как показано в следующем примере. (`ArrayRC` – это шаблон класса, также определенный в

```
template <class Type>
   class ArrayRC : public Array<Type> { /* ... */ };

int main() {
   ArrayRC<int> ia_rc(10);
   min4( ia_rc );
}
```

главе 2; наследование классов подробно рассматривается в главах 17 и 18.)

Фактический аргумент `ia_rc` имеет тип `ArrayRC<int>`. Он не совпадает с типом формального параметра `Array<Type>&`. Но одним из базовых классов для `ArrayRC<int>` является `Array<int>`, так как он конкретизирован из шаблона класса, указанного в качестве формального параметра функции. Поскольку фактический аргумент является производным классом, то его можно использовать при выводе аргументов шаблона. Таким образом, перед выводом аргумент функции `ArrayRC<int>` преобразуется в тип `Array<int>`, после чего для аргумента шаблона `Type` выводится тип `int` и конкретизируется функция `min4(Array<int>&)`.

В процессе вывода одного аргумента шаблона могут принимать участие несколько аргументов функции. Если параметр шаблона встречается в списке параметров функции более одного раза, то каждый выведенный тип должен точно соответствовать типу,

```
template <class T> T min5( T, T ) { /* ... */ }
unsigned int ui;

int main() {
   // ошибка: нельзя конкретизировать min5( unsigned int, int )
   // должно быть: min5( unsigned int, unsigned int ) или
   //              min5( int, int )
   min5( ui, 1024 );
}
```

выведенному для того же аргумента шаблона в первый раз:

Оба фактических аргумента функции должны иметь один и тот же тип: либо `int`, либо `unsigned int`, поскольку в шаблоне они принадлежат к одному типу `T`. Аргумент

шаблона T, выведенный из первого аргумента функции, – это int. Аргумент же шаблона T, выведенный из второго аргумента функции, – это unsigned int. Поскольку они оказались разными, процесс вывода завершается неудачей и при конкретизации шаблона выдается сообщение об ошибке. (Избежать ее можно, если явно задать аргументы шаблона при вызове функции min5(). В разделе 10.4 мы увидим, как это делается.)

Ограничение на допустимые типы преобразований относится только к тем фактическим параметрам функции, которые принимают участие в выводе аргументов шаблона. К остальным аргументам могут применяться любые трансформации. В следующем шаблоне функции sum() есть два формальных параметра. Фактический аргумент op1 для первого параметра участвует в выводе аргумента Type шаблона, а второй фактический аргумент

```
template <class Type>
```

op2 – нет.

```
   Type sum( Type op1, int op2 ) { /* ... */ }
```

Поэтому при конкретизации шаблона функции sum() его можно подвергать любым трансформациям. (Преобразования типов, применимые к фактическим аргументам

```
int ai[] = { ... };
double dd;
int main() {
   // конкретизируется sum( int, int )
   sum( ai[0], dd );
```

функции, описываются в разделе 9.3.) Например:

```
}
```

Тип второго фактического аргумента функции dd не соответствует типу формального параметра int. Но это не мешает конкретизировать шаблон функции sum(), поскольку тип второго аргумента фиксирован и не зависит от параметров шаблона. Для этого вызова конкретизируется функция sum(int,int). Аргумент dd приводится к типу int с помощью преобразования целого типа в тип с плавающей точкой.

Таким образом, общий алгоритм вывода аргументов шаблона можно сформулировать следующим образом:

1. По очереди исследуется каждый фактический аргумент функции, чтобы выяснить, присутствует ли в соответствующем формальном параметре какой-нибудь параметр шаблона.
2. Если параметр шаблона найден, то путем анализа типа фактического аргумента выводится соответствующий аргумент шаблона.
3. Тип фактического аргумента функции не обязан точно соответствовать типу формального параметра. Для приведения типов могут быть применены следующие преобразования:
    - трансформации l-значения
    - преобразования спецификаторов

- приведение производного класса к базовому при условии, что формальный параметр функции имеет вид `T<args>&` или `T<args>*`, где список аргументов `args` содержит хотя бы один параметр шаблона.

4. Если один и тот же параметр шаблона найден в нескольких формальных параметрах функций, то аргумент шаблона, выведенный по каждому из соответствующих фактических аргументов, должен быть одним и тем же.

Упражнение 10.4

Назовите два типа преобразований, которые можно применять к фактическим аргументам функций, участвующим в процессе вывода аргументов шаблона.

Упражнение 10.5

```
template <class Type>
   Type min3( const Type* array, int size ) { /* ... */ }
template <class Type>
```

Пусть даны следующие определения шаблонов:

```
   Type min5( Type p1, Type p2 ) { /* ... */ }
```

```
double dobj1, dobj2;
float fobj1, fobj2;
char cobj1, cobj2;
int ai[5] = { 511, 16, 8, 63, 34 };

(a) min5( cobj2, 'c' );
(b) min5( dobj1, fobj1 );
```

Какие из приведенных ниже вызовов ошибочны? Почему?

```
(c) min3( ai, cobj1 );
```

10.4. Явное задание аргументов шаблона A

В некоторых ситуациях автоматически вывести типы аргументов шаблона невозможно. Как мы видели на примере шаблона функции `min5()`, если процесс вывода дает два различных типа для одного и того же параметра шаблона, то компилятор сообщает об ошибке – неудачном выводе аргументов.

В таких ситуациях приходится подавлять механизм вывода и задавать аргументы *явно*, указывая их с помощью заключенного в угловые скобки списка разделенных запятыми значений, который следует после имени конкретизируемого шаблона функции. Например, если мы хотим задать тип `unsigned int` в качестве значения аргумента шаблона T в рассмотренном выше примере использования `min5()`, то нужно записать

```
// конкретизируется min5( unsigned int, unsigned int )
```

вызов конкретизируемого шаблона так:

```
min5< unsigned int >( ui, 1024 );
```

В этом случае список аргументов шаблона <unsigned int> явно задает их типы. Поскольку аргумент шаблона теперь известен, вызов функции больше не приводит к ошибке.

Обратите внимание, что при вызове функции min5() второй аргумент равен 1024, т.е. имеет тип int. Так как тип второго формального параметра функции при явном задании аргумента шаблона установлен в unsigned int, то второй фактический параметр функции приводится к типу unsigned int с помощью стандартного преобразования целых типов.

В предыдущем разделе мы говорили, что в процессе вывода аргументов шаблона к фактическим аргументам функции разрешается применять только ограниченное множество преобразований типов. Трансформация int в unsigned int в это множество не входит. Но если аргументы шаблона задаются явно, выполнять вывод типов не нужно, поскольку они уже зафиксированы. Следовательно, при явном задании аргументов шаблона для приведения типов фактических аргументов функции к типам формальных параметров можно применять любые стандартные преобразования.

Помимо разрешения любых преобразований фактических аргументов функции, явное задание аргументов шаблона помогает избежать и других проблем, встающих перед программистом. Рассмотрим следующую задачу. Мы хотим определить шаблон функции с именем sum() так, чтобы его конкретизация возвращала значения типа, достаточно большого для представления суммы двух значений любых двух типов, переданных в

```
// каким должен быть тип возвращаемого значения: T или U
template <class T, class U>
```

любом порядке. Как это сделать? Какой тип возвращаемого значения следует задать?

```
??? sum( T, U );
```

В нашем случае нельзя использовать ни тот, ни другой параметрический тип, иначе мы

```
char ch; unsigned int ui;

// ни T, ни U нельзя использовать в качестве типа возвращаемого значения
sum( ch, ui );   // правильно: U sum( T, U );
```

неизбежно допустим ошибку:

```
sum( ui, ch );   // правильно: T sum( T, U );
```

Решение заключается в том, чтобы ввести в шаблон третий параметр для обозначения

```
// T1 не появляется в списке параметров шаблона функции
template <class T1, class T2, class T3>
```

типа возвращаемого значения:

```
T1 sum( T2, T3 );
```

Поскольку тип возвращаемого значения может отличаться от типов аргументов функции, T1 не упоминается в списке формальных параметров. Это потенциальная проблема, так как тип T1 не может быть выведен из фактических аргументов функции. Однако, если

при конкретизации sum() мы зададим аргументы шаблона явно, то избегнем сообщения

```
typedef unsigned int ui_type;
ui_type calc( char ch, ui_type ui ) {

   // ...
   // ошибка: невозможно вывести T1
   ui_type loc1 = sum( ch, ui );

   // правильно: аргументы шаблона заданы явно
   // T1 и T3 - это unsigned int, T2 - это char
   ui_type loc2 = sum< ui_type, ui_type >( ch, ui );
}
```

компилятора о невозможности вывести T1. Например:

Не хватает возможности явно задать T1, но не T2 и T3, поскольку их можно вывести из аргументов функции при вызове.

При явном задании аргументов шаблона необходимо перечислять только те, которые не могут быть выведены автоматически. Но, как и в случае аргументов функции со

```
// правильно: T3 - это unsigned int
// T3 выведен из типа ui
ui_type loc3 = sum< ui_type, char >( ch, ui );

// правильно: T2 - это char, T3 - unsigned int
// T2 и T3 выведены из типа pf
ui_type (*pf)( char, ui_type ) = &sum< ui_type >;

// ошибка: опускать можно только “хвостовые” аргументы
```

значениями по умолчанию, опускать можно исключительно “хвостовые”:

```
ui_type loc4 = sum< ui_type, , ui_type >( ch, ui );
```

Встречаются ситуации, когда невозможно вывести аргументы шаблона в контексте, где конкретизируется шаблон функции; следовательно, необходимо их явно задать. Именно выявление таких ситуаций и необходимость решить проблему послужила причиной поддержки явного задания аргументов шаблона в стандартном C++.

В следующем примере берется адрес конкретизированной функции sum() и передается в качестве аргумента перегруженной функции manipulate(). Как мы показали в разделе 10.2, невозможно понять, как именно нужно конкретизировать sum(), если есть только списки параметров функций manipulate(). Имеется две разных функции sum(), и обе удовлетворяют условиям вызова. Следовательно, вызов manipulate() *неоднозначен*. Одним из способов разрешения такой неоднозначности является явное приведение типов. Однако лучше использовать явное задание аргументов шаблона: оно позволяет указать, как именно конкретизировать sum(), и, следовательно, выбрать нужный вариант перегруженной функции manipulate(). Например:

```
template <class T1, class T2, class T3>
   T1 sum( T2 op1, T3 op2 ) { /* ... */ }

void manipulate( int (*pf)( int,char ) );
void manipulate( double (*pf)( float,float ) );

int main()
{
   // ошибка: какой из возможных экземпляров sum:
   // int sum( int,char ) или double sum( float, float )?
   manipulate( &sum );

   // берется адрес конкретизированного экземпляра
   // double sum( float, float )
   // вызывается: void manipulate( double (*pf)( float, float ) );
   manipulate( &sum< double, float, float > );

}
```

Отметим, что явное задание аргументов шаблона следует использовать только тогда, когда это абсолютно необходимо для разрешения неоднозначности или для конкретизации шаблона функции в контексте, где вывести аргументы невозможно. Во-первых, определение типов и значений аргументов шаблона проще оставить компилятору. А во-вторых, если мы модифицируем объявления в программе, так что типы аргументов функции при вызове конкретизированного шаблона изменятся, то компилятор автоматически скорректирует вызов без нашего вмешательства. С другой стороны, если аргументы шаблона заданы явно, необходимо проверить, что они по-прежнему отвечают новым типам аргументов функции. Поэтому мы рекомендуем избегать явного задания аргументов шаблона.

Упражнение 10.6

Назовите две ситуации, когда использование явного задания аргументов шаблона необходимо.

Упражнение 10.7

```
template <class T1, class T2, class T3>
```

Пусть дано следующее определение шаблона функции sum():

```
   T1 sum( T2, T3 );
```

```
double dobj1, dobj2;
float fobj1, fobj2;
char cobj1, cobj2;

(a) sum( dobj1, dobj2 );
(b) sum<double,double,double>( fobj1, fobj2 );
(c) sum<int>( cobj1, cobj2 );
```

Какие из приведенных ниже вызовов ошибочны? Почему?

```
(d) sum<double, ,double>( fobj2, dobj2 );
```

10.5. Модели компиляции шаблонов A

Шаблон функции задает алгоритм для построения определений множества экземпляров функций. Сам шаблон не определяет никакой функции. Например, когда компилятор

```
template <typename Type>
   Type min( Type t1, Type t2 )
{
   return t1 < t2 ? t1 : t2;
}
```

видит шаблон:

```
int i, j;
```

он сохраняет внутреннее представление `min()`, но и только. Позже, когда встретится ее

```
double dobj = min( i, j );
```

реальное использование, скажем:

компилятор строит определение `min()` по сохраненному внутреннему представлению.

Здесь возникает несколько вопросов. Чтобы компилятор мог конкретизировать шаблон функции, должно ли его определение быть видимо при вызове экземпляра этой функции? Например, нужно ли определению шаблона `min()` появиться до ее конкретизации с целыми параметрами при инициализации `dobj`? Следует ли помещать шаблоны в заголовочные файлы, как мы поступаем с определениями встроенных (inline) функций? Или в заголовочные файлы можно помещать только объявления шаблонов, оставляя определения в файлах исходных текстов?

Чтобы ответить на эти вопросы, нам придется объяснить принятую в C++ *модель компиляции шаблонов*, сформулировать требования к организации определений и объявлений шаблонов в программах. В C++ поддерживаются две таких модели: модель с включением и модель с разделением. В данном разделе описываются обе модели и объясняется их использование.

10.5.1. Модель компиляции с включением

Согласно этой модели мы включаем определение шаблона в каждый файл, где этот шаблон конкретизируется. Обычно оно помещается в заголовочный файл, как и для

```
// model1.h
// модель с включением:
// определения шаблонов помещаются в заголовочный файл

template <typename Type>
   Type min( Type t1, Type t2 ) {

   return t1 < t2 ? t1 : t2;
}
```

встроенных функций. Именно такой моделью мы пользуемся в нашей книге. Например:

Этот заголовочный файл включается в каждый файл, где конкретизируется функция

```
// определения шаблонов включены раньше
// используется конкретизация шаблона
#include "model1.h"

int i, j;
```

`min()`:

```
double dobj = min( i, j );
```

Заголовочный файл можно включить в несколько файлов с исходными текстами программы. Означает ли это, что компилятор конкретизирует экземпляр функции `min()` с целыми параметрами в каждом файле, где имеется обращение к ней? Нет. Программа должна вести себя так, словно `min()` с целыми параметрами определена только один раз. Где и когда в действительности конкретизируется шаблон функции, оставляется на усмотрение разработчика компилятора. Нам достаточно знать, что где-то в программе нужная функция `min()` была конкретизирована. (Как мы покажем далее, с помощью явного объявления конкретизации можно указать, где и когда оно должно быть выполнено. Такие объявления желательно использовать на поздних стадиях разработки продукта для улучшения производительности.)

Решение включать определения шаблонов функций в заголовочные файлы не всегда удачно. Тело шаблона описывает детали реализации, которые пользователям не интересны или которые мы хотели бы от них скрыть. В действительности, если определение шаблона велико, то количество кода в заголовочном файле может превысить разумные пределы. Кроме того, многократная компиляция одного и того же определения при обработке разных файлов увеличивает общее время компиляции программы. Отделить объявления шаблонов функций от их определений позволяет модель компиляции с разделением. Посмотрим, как ее можно использовать.

10.5.2. Модель компиляции с разделением

Согласно этой модели объявления шаблонов функций помещаются в заголовочный файл, а определения – в файл с исходным текстом программы, т.е. объявления и определения шаблонов организованы так же, как в случае с невстроенными (non-inline) функциями.

```
// model2.h
// модель с разделением
// сюда помещается только объявление шаблона

template <typename Type> Type min( Type t1, Type t2 );

// model2.C
// определение шаблона
export template <typename Type>
```

Например:

```
    Type min( Type t1, Type t2 ) { /* ... */ }
```

Программа, которая конкретизирует шаблон функции `min()`, должна предварительно включить этот заголовочный файл:

```
// user.C
#include "model2.h"

int i, j;

double d = min ( i, j ); // правильно: здесь производится конкретизация
```

Хотя определение шаблона функции `min()` не видно в файле `user.c`, конкретизацию `min(int,int)` произвести можно. Но для этого шаблон `min()` должен быть определен специальным образом. Вы уже заметили, как именно? Если вы внимательно посмотрите на файл `model2.c`, то увидите, что определению шаблона функции `min()` предшествует ключевое слово `export`. Таким образом, шаблон `min()` становится э*кспортируемым*. Слово `export` говорит компилятору, что данное определение шаблона может понадобиться для конкретизации функций в других файлах. В таком случае компилятор должен гарантировать, что это определение будет доступно во время конкретизации.

Для объявления экспортируемого шаблона перед ключевым словом `template` в его определении надо поместить слово `export`. Если шаблон экспортируется, то его разрешается конкретизировать в любом исходном файле программы – для этого нужно лишь объявить его перед использованием. Если слово `export` перед определением опущено, то компилятор может и не конкретизировать экземпляр функции `min()` с целыми параметрами и нам не удастся связать программу.

Обратите внимание, что в некоторых реализациях это ключевое слово не нужно, поскольку поддерживается расширение языка, согласно которому неэкспортированный шаблон функции может встречаться только в одном исходном файле, при этом экземпляры такого шаблона в других файлах конкретизируются правильно. Однако подобное поведение не соответствует стандарту, который требует, чтобы пользователь всегда помечал определения шаблонов функций как экспортируемые, если объявление шаблона видно в исходном файле до его конкретизации.

Ключевое слово `export` в объявлении шаблона, находящемся в заголовочном файле, можно опустить. Так, в объявлении `min()` в файле `model2.h` этого слова нет.

Шаблон функции должен быть определен как экспортируемый только один раз во всей программе. К сожалению, поскольку компилятор обрабатывает файлы один за другим, он обычно не замечает, что шаблон определен как экспортируемый в нескольких исходных файлах. В результате подобного недосмотра может произойти следующее:

- при редактировании связей возникает ошибка, показывающая, что шаблон функции определен более, чем в одном файле;
- компилятор несколько раз конкретизирует шаблон функции с одним и тем же множеством аргументов, что приводит к ошибке повторного определения функции при связывании программы;
- компилятор может конкретизировать шаблон с помощью одного из его экспортированных определений, игнорируя все остальные.

Нельзя с уверенностью утверждать, что наличие в программе нескольких экспортируемых определений шаблона функции обязательно вызовет ошибку. При организации программы надо быть внимательным и следить за тем, чтобы подобные определения размещались только в одном исходном файле.

Модель с разделением позволяет отделить интерфейс шаблонов функций от его реализации и организовать программу так, что интерфейсы всех шаблонов помещаются в заголовочные файлы, а реализации – в файлы с исходным текстом. Однако не все

компиляторы поддерживают такую модель, а те, которые поддерживают, не всегда делают это правильно: модель с разделением требует более изощренной среды программирования, которая доступна не во всех реализациях C++. (В другой нашей книге, “Inside C++ Object Model”, описан механизм конкретизации шаблонов, поддержанный в одной из реализаций C++, а именно в компиляторе Edison Design Group.)

Поскольку приводимые нами примеры работы с шаблонами невелики и поскольку мы хотим, чтобы они компилировались максимально большим числом компиляторов, мы ограничились использованием модели с включением.

10.5.3. Явные объявления конкретизации

При использовании модели с включением определение шаблона функций включается в каждый исходный файл, где встречается конкретизация этого шаблона. Мы отмечали, что, хотя неизвестно, где и когда понадобится шаблон функции, программа должна вести себя так, как будто экземпляр шаблона для данного множества аргументов конкретизирован *ровно один раз*. В действительности некоторые компиляторы (особенно старые) конкретизируют шаблон функции с данным множеством аргументов шаблона неоднократно. В рамках этой модели для использования на этапе сборки или на одной из предшествующих ей стадий выбирается один из конкретизированных экземпляров, а остальные игнорируются.

Результат работы программы не зависит от того, сколько раз конкретизировался шаблон: в конечном итоге используется лишь один экземпляр. Но если приложение состоит из большого числа файлов, то время компиляции приложения заметно возрастает.

Подобные проблемы, характерные для старых компиляторов, затрудняли использование шаблонов. Поэтому в стандарте C++ введено понятие *явного объявления конкретизации*, помогающее программисту управлять моментом, когда конкретизация происходит.

В явном объявлении конкретизации за ключевым словом `template` идет объявление шаблона функции, в котором его аргументы указаны явно. Рассмотрим шаблон

```
template <typename Type>
   Type sum( Type op1, Type op2 ) { /* ... */ }

// явное объявление конкретизации
sum(int*, int):
template int* sum< int* >( int*, int );
```

Здесь в качестве аргумента явно задается `int*`. Явное объявление конкретизации с одним и тем же множеством аргументов шаблона может встречаться в программе не более одного раза.

Определение шаблона функции должно находиться в том же файле, где и явное объявление конкретизации. Если же его не видно, то явное объявление приводит к ошибке:

```
#include <vector>

template <typename Type>
   Type sum( Type op1, int op2 );  // только объявление

// определяем typedef для vector< int >
typedef vector< int > VI;

// ошибка: sum() не определен
template VI sum< VI >( VI , int );
```

Если в некотором исходном файле встречается явное объявление конкретизации, то что произойдет в других файлах, где используется такая же конкретизация шаблона функции? Как сказать компилятору, что явное объявление находится в другом файле и что при использовании в этом файле шаблон конкретизировать не надо?

Явные объявления конкретизации используются в сочетании с опцией компилятора, которая подавляет неявную конкретизацию шаблонов. Название опции в разных компиляторах различно. Например, в VisualAge for C++ для Windows версии 3.5 фирмы IBM эта опция называется `/ft-`. Если приложение компилируется с данной опцией, то компилятор предполагает, что шаблоны будут конкретизироваться явно, и не выполняет автоматической конкретизации.

Разумеется, если мы не включили в программу явного объявления конкретизации для некоторого шаблона, но задали опцию `/ft-`, то при сборке произойдет ошибка из-за того, что функция не была конкретизирована.

Упражнение 10.8

Назовите две модели компиляции шаблонов, поддерживаемые в C++. Объясните, как организуются определения шаблонов функций в каждой модели.

Упражнение 10.9

```
template <typename Type>
```

Пусть дано следующее определение шаблона функции `sum()`:

```
   Type sum( Type op1, char op2 );
```

Как записать явное объявление конкретизации этого шаблона с аргументом типа `string`?

10.6. Явная специализация шаблона A

Не всегда удается написать шаблон функции, который годился бы для всех возможных типов, с которыми он может быть конкретизирован. В некоторых случаях имеется специальная информация о типе, позволяющая написать более эффективную функцию, чем конкретизированная по шаблону. А иногда общее определение, предоставляемое шаблоном, для некоторого типа просто не работает. Рассмотрим, например, следующее определение шаблона функции `max()`:

```
// обобщенное определение шаблона
template <class T>
   T max( T t1, T t2 ) {
      return ( t1 > t2 ? t1 : t2 );
}
```

Когда этот шаблон конкретизируется с аргументом типа `const char*`, то обобщенное определение оказывается семантически некорректным, если мы интерпретируем каждый аргумент как строку символов в смысле языка C, а не как указатель на символ. В этом случае необходимо предоставить специализированное определение для конкретизации шаблона.

Явное определение специализации – это такое определение, в котором за ключевым словом `template` следует пара угловых скобок `<>`, а за ними – определение специализированного шаблона. Здесь указывается имя шаблона, аргументы, для которых он специализируется, список параметров функции и ее тело. В следующем примере для

```
#include <cstring>

// явная специализация для const char*:
// имеет приоритет над конкретизацией шаблона
// по обобщенному определению

typedef const char *PCC;
template<> PCC max< PCC >( PCC s1, PCC s2 ) {
```

`max(const char*, const char*)` определена явная специализация:

```
   return ( strcmp( s1, s2 ) > 0 ? s1 : s2 );
```

Поскольку имеется явная специализация, шаблон не будет конкретизирован с типом `const char*` при вызове в программе функции `max(const char*, const char*)`. При любом обращении к `max()` с двумя аргументами типа `const char*` работает специализированное определение. Для любых других обращений функция сначала конкретизируется по обобщенному определению шаблона, а затем вызывается. Вот как

```
#include <iostream>

// здесь должно быть определение шаблона функции max()
// и его специализации для аргументов const char*

int main() {
   // вызов конкретизированной функции: int max< int >( int, int );
   int i = max( 10, 5 );

   // вызов явной специализации:
   // const char* max< const char* >( const char*, const char* );
   const char *p = max( "hello", "world" );

   cout << "i: " << i << " p: " << p << endl;
   return 0;
```

это выглядит:

```
}
```

Можно объявлять явную специализацию шаблона функции, не определяя ее. Например,

```
// объявление явной специализации шаблона функции
```

для функции `max(const char*, const char*)` она объявляется так:

```
template< > PCC max< PCC >( PCC, PCC );
```

При объявлении или определении явной специализации шаблона функции нельзя опускать слово `template` и следующую за ним пару скобок `<>`. Кроме того, в объявлении

```
// ошибка: неправильные объявления специализации

// отсутствует template<>
PCC max< PCC >( PCC, PCC );

// отсутствует список параметров
```

специализации обязательно должен быть список параметров функции:

```
template<> PCC max< PCC >;
```

Однако здесь можно опускать задание аргументов шаблона, если они выводятся из

```
// правильно: аргумент шаблона const char* выводится из типов параметров
```

формальных параметров функции:

```
template<> PCC max( PCC, PCC );
```

```
template <class T1, class T2, class T3>
   T1 sum( T2 op1, T3 op2 );

// объявления явных специализаций

// ошибка: аргумент шаблона для T1 не может быть выведен;
// он должен быть задан явно
template<> double sum( float, float );

// правильно: аргумент для T1 задан явно,
// T2 и T3 выводятся и оказываются равными float
template<> double sum<double>( float, float );

// правильно: все аргументы заданы явно
```

В следующем примере шаблон функции `sum()` явно специализирован:

```
template<> int sum<int,char>( char, char );
```

Пропуск части `template<>` в объявлении явной специализации не всегда является ошибкой. Например:

```
// обобщенное определение шаблона
template <class T>
   T max( T t1, T t2 ) { /* ... */ }

// правильно: обычное объявление функции
const char* max( const char*, const char*);
```

Однако эта инструкция не является специализацией шаблона функции. Здесь просто объявляется обычная функция с типом возвращаемого значения и списком параметров, которые соответствуют полученным при конкретизации шаблона. Объявление обычной функции, являющееся конкретизацией шаблона, не считается ошибкой.

Так почему бы просто не объявить обычную функцию? Как было показано в разделе 10.3, для преобразования фактического аргумента функции, конкретизированной по шаблону, в соответствующий формальный параметр в случае, когда этот аргумент принимает участие в выводе аргумента шаблона, может быть применено лишь ограниченное множество преобразований типов. Точно так же обстоит дело и в ситуации, когда шаблон функции специализируется явно: к фактическим аргументам функции при этом тоже применимо лишь ограниченное множество преобразований. Явные специализации не помогают обойти соответствующие ограничения. Если мы хотим выйти за их пределы, то должны определить обычную функцию вместо специализации шаблона. (В разделе 10.8 этот вопрос рассматривается более подробно; там же показано, как работает разрешение перегруженной функции для вызова, который соответствует как обычной функции, так и экземпляру, конкретизированному из шаблона.)

Явную специализацию можно объявлять даже тогда, когда специализируемый шаблон объявлен, но не определен. В предыдущем примере шаблон функции `sum()` лишь объявлен к моменту специализации. Хотя определение шаблона не обязательно, объявление все же требуется. То, что `sum()` – шаблон, должно быть известно до того, как это имя может быть специализировано.

Такое объявление должно быть видимо до его использования в исходном файле.

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

// обобщенное определение шаблона
template <class T>
   T max( T t1, T t2 ) { /* ... */ }

int main() {
   // конкретизация функции
   // const char* max< const char* >( const char*, const char* );
   const char *p = max( "hello", "world" );

   cout << "p: " << p << endl;
   return 0;
}

// некорректная программа: явная специализация const char *:
// имеет приоритет над обобщенным определением шаблона
typedef const char *PCC;
```

Например:

```
template<> PCC max< PCC >(PCC s1, PCC s2 ) { /* ... */ }
```

В предыдущем примере конкретизация `max(const char*, const char*)` предшествует объявлению явной специализации. Поэтому компилятор имеет право предположить, что функция должна быть конкретизирована по обобщенному определению шаблона. Однако в программе не может одновременно существовать явная специализация и экземпляр, конкретизированный по тому же шаблону с тем же множеством аргументов. Когда в исходном файле после конкретизации встречается явная специализация `max(const char*, const char*)`, компилятор выдает сообщение об ошибке.

Если программа состоит из нескольких файлов, то объявление явной специализации шаблона должно быть видимо в каждом файле, в котором она используется. Не разрешается в одних файлах конкретизировать шаблон функции по обобщенному определению, а в других специализировать с тем же множеством аргументов. Рассмотрим

```
// --------- max.h -------
// обобщенное определение шаблона
template <class Type>
   Type max( Type t1, Type t2 ) { /* ... */ }

// --------- File1.C -------
#include <iostream>
#include "max.h"
void another();

int main() {
   // конкретизация функции
   // const char* max< const char* >( const char*, const char* );
   const char *p = max( "hello", "world" );

   cout << "p: " << p << endl;
   another();

   return 0;
}

// --------- File2.C -------
#include <iostream>
#include <cstring>
#include "max.h"

// явная специализация шаблона для const char*
typedef const char *PCC;
template<> PCC max< PCC >( PCC s1, PCC s2 ) { /* ... */ }

void another() {

   // явная специализация
   // const char* max< const char* >( const char*, const char* );
   const char *p = max( "hi", "again" );

   cout << " p: " << p << endl;

   return 0;
}
```

следующий пример:

Эта программа состоит из двух файлов. В файле `File1.C` нет объявления явной специализации `max(const char*, const char*)`. Вместо этого шаблон функции конкретизируется из обобщенного определения. В файле `File2.C` объявлена явная

специализация, и при обращении к `max("hi", "again")` именно она и вызывается. Поскольку в одной и той же программе функция `max(const char*, const char*)` то конкретизируется по шаблону, то специализируется явно, компилятор считает программу некорректной. Для исправления этого объявление явной специализации шаблона должно предшествовать вызову функции `max(const char*, const char*)` в файле `File1.C`.

Чтобы избежать таких ошибок и гарантировать, что объявление явной специализации шаблона `max(const char*, const char*)` внесено в каждый файл, где используется шаблон функции `max()` с аргументами типа `const char*`, это объявление следует поместить в заголовочный файл `"max.h"` и включать его во все исходные файлы, в

```
// --------- max.h -------
// обобщенное определение шаблона
template <class Type>
   Type max( Type t1, Type t2 ) { /* ... */ }

// объявление явной специализации шаблона для const char*
typedef const char *PCC;
template<> PCC max< PCC >( PCC s1, PCC s2 );

// --------- File1.C -------
#include <iostream>
#include "max.h"
void another();

int main() {
   // специализация
   // const char* max< const char* >( const char*, const char* );
   const char *p = max( "hello", "world" );

   // ....
```

которых используется шаблон `max()`:

```
}
```

Упражнение 10.10

Определите шаблон функции `count()` для подсчета числа появлений некоторого значения в массиве. Напишите вызывающую программу. Последовательно передайте в ней массив значений типа `double`, `int` и `char`. Напишите специализированный экземпляр шаблона `count()` для обработки строк.

10.7. Перегрузка шаблонов функций A

Шаблон функции может быть перегружен. В следующем примере есть три перегруженных объявления для шаблона `min()`:

```
// определение шаблона класса Array
// (см. раздел 2.4)

template <typename Type>
   class Array( /* ... */ };

// три объявления шаблона функции min()

template <typename Type>
   Type min( const Array<Type>&, int ); // #1

template <typename Type>
   Type min( const Type*, int ); // #2

template <typename Type>

   Type min( Type, Type ); // #3
```

Следующее определение `main()` иллюстрирует, как могут вызываться три объявленных

```
#include <cmath>

int main()
{
   Array<int> iA(1024);   // конкретизация класса
   int ia[1024];

   // Type == int; min( const Array<int>&, int )
   int ival0 = min( iA, 1024 );

   // Type == int; min( const int*, int )
   int ival1 = min( ia, 1024 );

   // Type == double; min( double, double )
   double dval0 = min( sqrt( iA[0] ), sqrt( ia[0] ) );

   return 0;
}
```

таким образом функции:

Разумеется, тот факт, что три перегруженных шаблона функции успешно объявлены, не означает, что они могут быть также успешно вызваны. Такие шаблоны могут приводить к неоднозначности при вызове конкретизированного шаблона. Например, для следующего

```
template <typename T>
```

определения шаблона `min5()`

```
   int min5( T, T ) { /* ... */ }
```

функция не конкретизируется по шаблону, если `min5()` вызывается с аргументами разных типов; при этом процесс вывода заканчивается с ошибкой, поскольку из фактических аргументов функции выводятся два разных типа для `T`.

```
int i;
unsigned int ui;

// правильно: для T выведен тип int
min5( 1024, i );

// вывод аргументов шаблона заканчивается с ошибкой:
// для T можно вывести два разных типа
min5 ( i, ui );
```

Для разрешения второго вызова можно было бы перегрузить min5(), допустив два различных типа аргументов:

```
template <typename T, typename U>
   int min5( T, U );

// правильно: int min5( int, usigned int )
```

При следующем обращении производится конкретизация этого шаблона функции:

```
min5( i, ui );

// ошибка: неоднозначность: две возможных конкретизации
//         из min5( T, T ) и min5( T, U )
```

К сожалению, теперь стал неоднозначным предыдущий вызов:

```
min5( 1024, i );
```

Второе объявление min5() допускает наличие у функции аргументов различных типов, но не требует этого. В нашем случае и T, и U типа int. Оба объявления шаблонов могут быть конкретизированы вызовом, в котором два аргумента функции имеют один и тот же тип. Единственный способ указать, какой шаблон более предпочтителен, устранив тем самым неоднозначность, – явно задать его аргументы. (О явном задании аргументов шаблона см. раздел 10.4.) Например:

```
// правильно: конкретизация из min5( T, U )
min5<int, int>( 1024, i );
```

Однако в этом случае мы можем обойтись без перегрузки шаблона функции. Поскольку шаблон min5(T,U) подходит для всех вызовов, для которых подходит min5(T,T), то одного объявления min5(T,U) вполне достаточно, а объявление min5(T,T) можно удалить. Мы уже говорили в главе 9, что, хотя перегрузка допускается, при проектировании таких функций надо быть внимательным и использовать ее только при необходимости. Те же соображения применимы и к определению перегруженных шаблонов.

В некоторых ситуациях неоднозначности при вызове не возникает, хотя по шаблону можно конкретизировать две разных функции. Если имеются следующие два шаблона для функции sum(), то предпочтение будет отдано первому даже тогда, когда

```
template <typename Type>
   Type sum( Type*, int );

template <typename Type>
   Type sum( Type, int );

int ia[1024];

// Type == int ; sum<int>( int*, int ); или
// Type == int*; sum<int*>( int*, int ); ??
```

конкретизированы могут быть оба:

```
int ival1 = sum<int>( ia, 1024 );
```

Как это ни удивительно, такой вызов не приводит к неоднозначности. Шаблон конкретизируется из первого определения, так как выбирается *наиболее специализированное* определение. Поэтому для аргумента Type принимается int, а не int*.

Для того чтобы один шаблон был более специализирован, чем другой, оба они должны иметь одни и те же имя и число параметров, а для параметров разных типов, как, скажем, T* и T в предыдущем примере, параметр в одном шаблоне должен быть способен принять более широкое множество фактических аргументов, чем соответствующий параметр в другом. Например, для шаблона sum(Type*, int) вместо первого формального параметра функции разрешается подставлять только фактические аргументы типа “указатель”. В то же время в шаблоне sum(Type, int) первому формальному параметру могут соответствовать фактические аргументы любого типа. Первый шаблон sum(Type*, int) допускает более узкое множество аргументов, чем второй, т.е. он более специализирован, а следовательно, он и конкретизируется при вызове функции.

10.8. Разрешение перегрузки при конкретизации A

В предыдущем разделе мы видели, что шаблон функции может быть перегружен. Кроме того, допускается использование одного и того же имени для шаблона и обычной

```
// шаблон функции
template <class Type>
   Type sum( Type, int ) { /* ... */ }
// обычная функция (не шаблон)
```

функции:

```
double sum( double, double );
```

Когда программа обращается к sum(), вызов разрешается либо в пользу конкретизированного экземпляра шаблона, либо в пользу обычной функции – это зависит от того, какая функция лучше соответствует фактическим аргументам. (Для решения такой проблемы применяется процесс разрешения перегрузки, описанный в главе 9.) Рассмотрим следующий пример:

```
void calc( int ii, double dd ) {
   // что будет вызвано: конкретизированный экземпляр шаблона
   // или обычная функция?
   sum( dd, ii );
}
```

Будет ли при обращении к `sum(dd,ii)` вызвана функция, конкретизированная из шаблона, или обычная функция? Чтобы ответить на этот вопрос, выполним по шагам процедуру разрешения перегрузки. Первый шаг заключается в построении множества функций-кандидатов состоящего из одноименных вызванной функций, объявления которых видны в точке вызова.

Если существует шаблон функции и на основе фактических аргументов вызова из него может быть конкретизирована функция, то она будет являться кандидатом. Так ли это на самом деле, зависит от результата процесса вывода аргументов шаблона. (Этот процесс описан в разделе 10.3.) В предыдущем примере для вывода значения аргумента `Type` шаблона используется фактический аргумент функции `dd`. Тип выведенного аргумента оказывается равным `double`, и к множеству функций-кандидатов добавляется функция `sum(double, int)`. Таким образом, для данного вызова имеются два кандидата: конкретизированная из шаблона функция `sum(double, int)` и обычная функция `sum(double, double)`.

После того как функции, конкретизированные из шаблона, включены в множество кандидатов, процесс вывода аргументов шаблона продолжается как обычно.

Второй шаг процедуры разрешения перегрузки заключается в выборе устоявших функций из множества кандидатов. Напомним, что устоявшей называется функция, для которой существуют преобразования типов, приводящие каждый фактический аргумент функции к типу соответствующего формального параметра. (В разделе 9.3 описаны преобразования типов, применимые к фактическим аргументам функции.) Нужные трансформации существуют как для конкретизированной функции `sum(double, int)`, так и для обычной функции `sum(double, double)`. Следовательно, обе они являются устоявшими.

Проведем ранжирование преобразований типов, примененных к фактическим аргументам для выбора наилучшей из устоявших функций. В нашем примере оно происходит следующим образом:

Для конкретизированной из шаблона функции `sum(double, int)`:

- для первого фактического аргумента как сам этот аргумент, так и формальный параметр имеют тип `double`, т.е. мы видим точное соответствие;
- для второго фактического аргумента как сам аргумент, так и формальный параметр имеют тип `int`, т.е. снова точное соответствие.

Для обычной функции `sum(double, double)`:

- для первого фактического аргумента как сам этот аргумент, так и формальный параметр имеют тип `double` – точное соответствие;
- для второго фактического аргумента сам этот аргумент имеет тип `int`, а формальный параметр – тип `double`, т.е. необходимо стандартное преобразование между целым и плавающим типами.

Если рассматривать только первый аргумент, то обе функции одинаково хороши. Однако для второго аргумента конкретизированная из шаблона функция лучше. Поэтому наиболее подходящей (лучшей из устоявших) считается функция sum(double, int).

Функция, конкретизированная из шаблона, включается в множество кандидатов только тогда, когда процесс вывода аргументов завершается успешно. Неудачное завершение в данном случае не является ошибкой, но кандидатом функция считаться не будет.

```
// шаблон функции
template <class T>
```

Предположим, что шаблон функции sum() объявлен следующим образом:

```
   int sum( T*, int ) { ... }
```

Для описанного вызова функции вывод аргументов шаблона будет неудачным, так как фактический аргумент типа double не может соответствовать формальному параметру типа T*. Поскольку для данного вызова и данного шаблона конкретизировать функцию невозможно, в множество кандидатов ничего не добавляется, т.е. единственным его элементом останется обычная функция sum(double, double). Именно она вызывается при обращении, и ее второй фактический аргумент приводится к типу double.

А если вывод аргументов шаблона завершается удачно, но для них есть явная специализация? Тогда именно она, а не функция, конкретизированная из обобщенного

```
// определение шаблона функции
template <class Type> Type sum( Type, int ) { /* ... */ }

// явная специализация для Type == double
template<> double sum<double>( double,int );

// обычная функция
double sum( double, double );

void manip( int ii, double dd ) {
   // вызывается явная специализация шаблона sum<double>()
   sum( dd, ii );
```

шаблона, попадает в множество кандидатов. Например:

```
}
```

При обращении к sum() внутри manip() в процессе вывода аргументов шаблона обнаруживается, что функция sum(double,int), конкретизированная из обобщенного шаблона, должна быть добавлена к множеству кандидатов. Но для нее имеется явная специализация, которая и становится кандидатом. На более поздних стадиях анализа выясняется, что эта специализация дает наилучшее соответствие фактическим аргументам вызова, так что разрешение перегрузки завершается в ее пользу.

Явные специализации шаблона не включаются в множество кандидатов автоматически. Лишь в том случае, когда вывод аргументов завершается успешно, компилятор будет рассматривать явные специализации данного шаблона:

```
// определение шаблона функции
template <class Type>
   Type min( Type, Type ) { /* ... */ }

// явная специализация для Type == double
template<> double min<double>( double, double );

void manip( int ii, double dd ) {
   // ошибка: вывод аргументов шаблона неудачен,
   // нет функций-кандидатов для данного вызова
   min( dd, ii );
}
```

Шаблон функции `min()` специализирован для аргумента `double`. Однако эта специализация не попадает в множество функций-кандидатов. Процесс вывода для вызова `min()` завершился неудачно, поскольку аргументы шаблона, выведенные для `Type` на основе разных фактических аргументов функции, оказались различными: для первого аргумента выводится тип `double`, а для второго – `int`. Поскольку вывести аргументы не удалось, в множество кандидатов никакая функция не добавляется, и специализация `min(double, double)` игнорируется. Так как других функций-кандидатов нет, вызов считается ошибочным.

Как отмечалось в разделе 10.6, тип возвращаемого значения и список формальных параметров обычной функции может точно соответствовать аналогичным атрибутам функции, конкретизированной из шаблона. В следующем примере `min(int,int)` – это обычная функция, а не специализация шаблона `min()`, поскольку, как вы, вероятно,

```
// объявление шаблона функции
template <class T>
   T min( T, T );

// обычная функция min(int,int)
int min( int, int ) { }
```

помните, объявление специализации должно начинаться с `template<>`:

Вызов может точно соответствовать как обычной функции, так и функции, конкретизированной из шаблона. В следующем примере оба аргумента в `min(ai[0],99)` имеют тип `int`. Для этого вызова есть две устоявших функции: обычная `min(int,int)` и конкретизированная из шаблона функция с тем же типом возвращаемого значения и

```
int ai[4] = { 22, 33, 44, 55 };
int main() {
   // вызывается обычная функция min( int, int )
   min( ai[0], 99 );
}
```

списком параметров:

Однако такой вызов не является неоднозначным. Обычной функции, если она существует, всегда отдается предпочтение, поскольку она реализована явно, так что перегрузка разрешается в пользу обычной функции `min(int,int)`.

Если перегрузка разрешилась таким образом, то изменений уже не будет: если позже обнаружится, что в программе нет определения этой функции, компилятор не станет конкретизировать ее тело из шаблона. Вместо этого на этапе сборки мы получим ошибку. В следующем примере программа вызывает, но не определяет обычную функцию

```
// шаблон функции
template <class T>
   T min( T, T ) { ... }

// это обычная функция, не определенная в программе
int min( int, int );

int ai[4] = { 22, 33, 44, 55 };
int main() {
   // ошибка сборки: min( int, int ) не определена
   min( ai[0], 99 );
```

`min(int,int)`, и редактор связей выдает сообщение об ошибке:

```
}
```

Зачем определять обычную функцию, если ее тип возвращаемого значения и список параметров соответствуют функции, конкретизированной из шаблона? Вспомните, что при вызове конкретизированной функции к ее фактическим аргументам в ходе вывода аргументов шаблона можно применять только ограниченное множество преобразований. Если же объявлена обычная функция, то для приведения типов аргументов допустимы любые трансформации, так как типы формальных параметров обычной функции фиксированы. Рассмотрим пример, показывающий, зачем может потребоваться объявить обычную функцию.

Предположим, что мы хотим определить специализацию шаблона функции `min<int>(int,int)`. Нужно, чтобы именно эта функция вызывалась при обращении к `min()` с аргументами любых целых типов, пусть даже неодинаковых. Из-за ограничений, наложенных на преобразования типов, при передаче фактических аргументов разных типов функция `min<int>(int,int)` не будет конкретизирована из шаблона. Мы могли бы заставить компилятор выполнить конкретизацию, явно задав аргументы шаблона, однако решение, при котором не требуется модифицировать каждый вызов, предпочтительнее. Определив обычную функцию, мы добьемся того, что программа будет вызывать специальную версию `min(int,int)` для любых фактических аргументов целых типов без явного указания аргументов шаблона:

```
// определение шаблона функции
template <class Type>
   Type min( Type t1, Type t2 ) { ... }

int ai[4] = { 22, 33, 44, 55 };
short ss = 88;

void call_instantiation() {
   // ошибка: для этого вызова нет функции-кандидата
   min( ai[0], ss );
}

// обычная функция
int min( int a1, int a2 ) {
   min<int>( a1, a2 );
}

int main() {
   call_instantiation() {
   // вызывается обычная функция
   min( ai[0], ss );
}
```

Для вызова `min(ai[0],ss)` из `call_instantiation` нет ни одной функции-кандидата. Попытка сгенерировать ее из шаблона `min()` провалится, поскольку для аргумента шаблона `Type` из фактических аргументов функции выводятся два разных значения. Следовательно, такой вызов ошибочен. Однако при обращении к `min(ai[0],ss)` внутри `main()` видимо объявление обычной функции `min(int, int)`. Тип первого фактического аргумента этой функции точно соответствует типу формального параметра, а второй аргумент может быть преобразован в тип формального параметра с помощью расширения типа. Поскольку для второго вызова устояла только данная функция, то она и вызывается.

Разобравшись с разрешением перегрузки функций, конкретизированных из шаблонов, специализацией шаблонов функций и обычных функций с тем же именем, подытожим все, что мы об этом рассказали:

1. Построить множество функций-кандидатов.

   Рассматриваются шаблоны функций с тем же именем, что и вызванная. Если аргументы шаблона выведены из фактических аргументов функции успешно, то в множество функций-кандидатов включается либо конкретизированный шаблон, либо специализация шаблона для выведенных аргументов, если она существует.

2. Построить множество устоявших функций (см. раздел 9.3).

   В множестве функций-кандидатов остаются только функции, которые можно вызвать с данными фактическими аргументами.

3. Ранжировать преобразования типов (см. раздел 9.3).

   a. Если есть только одна функция, вызвать именно ее.

   b. Если вызов неоднозначен, удалить из множества устоявших функции, конкретизированные из шаблонов.

4. Разрешить перегрузку, рассматривая среди всех устоявших только обычные функции (см. раздел 9.3).

   a. Если есть только одна функция, вызвать именно ее.

b. В противном случае вызов неоднозначен.

Проиллюстрируем эти шаги на примере. Предположим, есть два объявления – шаблона

```
template <class Type>
   Type max( Type, Type ) { ... }

// обычная функция
```

функции и обычной функции. Оба принимают аргументы типа double:

```
double max( double, double );
```

А вот три вызова max(). Можете ли вы сказать, какая функция будет вызвана в каждом

```
int main() {
   int ival;
   double dval;
   float fd;

   // ival, dval и fd присваиваются значения

   max( 0, ival );
   max( 0.25, dval );
   max( 0, fd );
```

случае?

```
}
```

Рассмотрим последовательно все три вызова:

1. max(0,ival). Оба аргумента имеют тип int. Для вызова есть два кандидата: конкретизированная из шаблона функция max(int, int) и обычная функция max(double, double). Конкретизированная функция точно соответствует фактическим аргументам, поэтому она и вызывается;
2. max(0.25,double). Оба аргумента имеют тип double. Для вызова есть два кандидата: конкретизированная из шаблона max(double, double) и обычная max(double, double). Вызов неоднозначен, поскольку точно соответствует обеим функциям. Правило 3b говорит, что в таком случае выбирается обычная функция;.
3. max(0,fd). Аргументы имеют тип int и float соответственно. Для вызова существует только один кандидат: обычная функция max(double, double). Вывод аргументов шаблона заканчивается неудачей, так как значения типа Type, выведенные из разных фактических аргументов функции, различны. Поэтому в множество кандидатов конкретизированная из шаблона функция не попадает. Обычная же функция устояла, поскольку существуют преобразования типов фактических аргументов в типы формальных параметров; она и выбирается. Если бы обычная функция не была объявлена, вызов закончился бы ошибкой.

А если бы мы определили еще одну обычную функцию для max()? Например:

```
template <class T> T max( T, T ) { ... }

// две обычные функции
char max( char, char );

double max( double, double );


int main() {
   float fd;

   // в пользу какой функции разрешается вызов?
   max( 0, fd );
```

Будет ли в таком случае третий вызов разрешен по-другому? Да.

```
}
```

Правило 3b говорит, что, поскольку вызов неоднозначен, следует рассматривать только обычные функции. Ни одна из них не считается наилучшей из устоявших, так как преобразования типов фактических аргументов одинаково плохи: в обоих случаях для установления соответствия требуется стандартная трансформация. Таким образом, вызов неоднозначен, и компилятор сообщает об ошибке.

Упражнение 10.11

```
template <class Type>
   Type max( Type, Type ) { ... }
```

Вернемся к представленному ранее примеру:

```
int main() {
   int ival;
   double dval;
   float fd;

   max( 0, ival );
   max( 0.25, dval );
   max( 0, fd );

double max( double, double );
}
```

Добавим в множество объявлений в глобальной области видимости следующую специализацию шаблона функции:

```
template <> char max<char>* char, char ) { ... }
```

Составьте список кандидатов и устоявших функций для каждого вызова `max()` внутри `main()`.

Предположим, что в `main()` добавлен следующий вызов:

```
int main() {
   // ...
   max( 0, 'j' );
}
```

В пользу какой функции он будет разрешен? Почему?

Упражнение 10.12

Предположим, что есть следующее множество определений и специализаций шаблонов, а

```
int i;             unsigned int ui;
char str[24];      int ia[24];

template <class T> T calc( T*, int );
template <class T> T calc( T, T );
template<> chat calc( char*. int );
```

также объявления переменных и функций:

```
double calc( double, double );
```

Выясните, какая функция или конкретизированный шаблон вызывается в каждом из показанных ниже случаев. Для каждого вызова перечислите функции-кандидаты и

```
(a) cslc( str, 24 );        (d) calc( i, ui );
(b) calc( is, 24 );         (e) calc( ia, ui );
```

устоявшие функции; объясните, какая из устоявших функций будет наилучшей.

```
(c) calc( ia[0], 1 );       (f) calc( &i, i );
```

10.9. Разрешение имен в определениях шаблонов A

Внутри определения шаблона смысл некоторых конструкций может различаться в зависимости от конкретизации, тогда как смысл других всегда остается неизменным.

```
template <typename Type>
   Type min( Type* array, int size )
{
   Type min_val = array[0];
   for (int i = 1; i < size; ++i)
       if ( array[i] < min_val )
          min_val = array[i];
   print( "Minimum value found: ");
   print( min_val );

   return min_val;
}
```

Главную роль играет наличие в конструкции формального параметра шаблона:

В функции `min()` типы переменных `array` и `min_val` зависят от фактического типа, которым будет заменен `Type` при конкретизации шаблона, тогда как тип переменной

size останется int при любом типе параметра шаблона. Следовательно, типы array и min_val в разных конкретизациях различны. Поэтому мы говорим, что типы этих переменных *зависят от параметра шаблона*, тогда как тип size от него не зависит.

Так как тип min_val неизвестен, то неизвестна и операция, которая будет использоваться при появлении min_val в выражении. Например, какая функция print() будет вызвана при обращении print(min_val)? С типом аргумента int? Или float? Будет ли вызов ошибочным, поскольку не существует функции, которая может быть вызвана с аргументом того же типа, что и min_val? Принимая все это во внимание, мы говорим, что и вызов print(min_val) зависит от параметра шаблона.

Такие вопросы не возникают для тех конструкций внутри min(), которые не зависят от параметров шаблона. Например, всегда известно, какая функция должна быть вызвана для print("Minimum value found: "). Это функция печати строк символов. В данном случае print() остается одной и той же при любой конкретизации шаблона, то есть не зависит от его параметров.

В главе 7 мы видели, что в C++ функция должна быть объявлена до ее вызова. Нужно ли объявлять функцию, вызываемую внутри шаблона, до того, как компилятор увидит его определение? Должны ли мы объявить функцию print() в предыдущем примере до определения шаблона min()? Ответ зависит от особенностей имени, на которое мы ссылаемся. Конструкцию, не зависящую от параметров шаблона, следует объявить перед ее использованием в шаблоне. Представленное выше определение шаблона функции min() некорректно. Поскольку вызов

```
print( "Minimum value found: ");
```

не зависит от параметров шаблона, то функция print() для печати строк символов должна быть объявлена до использования. Чтобы исправить эту ошибку, можно

```
// ---- primer.h ----

// это объявление необходимо:
// внутри min() вызывается print( const char * )
void print( const char * );

template <typename Type>
   Type min( Type* array, int size ) {

   // ...

   print( "Minimum value found: ");
   print( min_val );

   return min_val;
}
```

поместить объявление print() перед определением min():

С другой стороны, объявление функции print(), используемой для печати min_val, пока не нужно, так как еще неизвестно, какую конкретно функцию надо искать. Мы не знаем, какая функция print() будет вызвана при обращении print(min_val), пока тип min_val не станет известным.

Когда же должна быть объявлена функция print(), вызываемая при обращении

```
#include <primer.h>
void print( int );

int ai[4] = {12, 8, 73, 45 };

int main() {
   int size = sizeof(ai) / sizeof(int);
   // конкретизируется min( int*, int )
   min( &ai[0], size );
```

print(min_val)? До конкретизации шаблона. Например:

```
}
```

main() вызывает конкретизированную из шаблона функцию min(int*,int). В этой реализации Type заменено int, и тип переменной min_val, следовательно, равен int. Поэтому при обращении print(min_val) вызывается функция с аргументом типа int. Именно тогда, когда конкретизируется min(int*,int), становится известно, что при втором вызове аргумент print() имеет тип int. В этот момент такая функция должна быть видима. Если бы функция print(int) не была объявлена до конкретизации min(int*,int), то компилятор выдал бы сообщение об ошибке.

Поэтому разрешение имен в определении шаблона происходит в два этапа. Сначала разрешаются имена, не зависящие от его параметров, а затем, при конкретизации, – имена, зависящие от параметров.

Но зачем нужны два шага? Почему бы, например, не разрешать все имена при конкретизации?

Если вы проектируете шаблон функции, то, вероятно, хотели бы сохранить контроль над тем, когда разрешаются имена в его определении. Предположим, что шаблон min() – это часть библиотеки, в которой определены и другие шаблоны и функции. Желательно, чтобы реализации min() по возможности использовали другие компоненты нашей же библиотеки. В предыдущем примере интерфейс библиотеки определен в заголовочном файле <primer.h>. Как объявление функции print(const char*), так и определение функции min() являются частями интерфейса. Мы хотим, чтобы конкретизации шаблона min() пользовались функцией print() из нашей библиотеки. Первый этап разрешения имени это гарантирует. Если имя, использованное в определении шаблона, не зависит от его параметров, то оно обязательно будет относиться к компоненту внутри библиотеки, т.е. к тому объявлению, которое включено в один пакет с этим определением в заголовочном файле <primer.h>.

На самом деле автор шаблона должен позаботиться о том, чтобы были объявлены все имена, использованные в определениях и не зависящие от параметров. Если этого нет, то определение шаблона вызовет ошибку. При конкретизации шаблона компилятор ее не исправляет:

```
// ---- primer.h ----
template <typename Type>
   Type min( Type* array, int size )
{
   Type min_val = array[0];
   // ...
   // ошибка: функция print( const char* ) не найдена
   print( "Minimum value found: " );

   // правильно: зависит от параметра шаблона
   print( min_val );
   // ...
}

// ---- user.C ----
#include <primer.h>

// это объявление print( const char* ) игнорируется
void print( const char* );
void print( int );

int ai[4] = {12, 8, 73, 45 };

int main() {
   int size = sizeof(ai) / sizeof(int);
   // конкретизируется min( int*, int )
   min( &ai[0], size );

}
```

Объявление функции `print( const char* )` в файле `user.C` невидимо в том месте, где появляется определение шаблона. Однако оно видимо там, где конкретизируется шаблон `min(int*,int)`, но это объявление не рассматривается при компиляции вызова `print("Minimum value found: ")`, так как последний не зависит от параметров шаблона. Если некоторая конструкция в определении шаблона не зависит от его параметров, то имена разрешаются в контексте самого определения, и результат разрешения в дальнейшем не пересматривается. Поэтому на программиста возлагается ответственность за то, чтобы объявления имен, встречающихся в определении, были включены в интерфейс библиотеки вместе с шаблоном.

А теперь предположим, что библиотека была написана кем-то другим, а мы ее пользователи, которым доступен интерфейс, определенный в заголовочном файле `<primer.h>`. Иногда нужно, чтобы объекты и функции, определенные в нашей программе, учитывались при конкретизации шаблона из библиотеки. Допустим, мы определили в своей программе класс `SmallInt` и хотели бы конкретизировать функцию `min()` из библиотеки `<primer.h>` для получения минимального значения в массиве объектов типа `SmallInt`.

При конкретизации шаблона `min()` для массива объектов типа `SmallInt` вместо аргумента шаблона `Type` подставляется тип `SmallInt`. Следовательно, `min_val` в конкретизированной функции `min()` имеет тот же тип. Тогда как разрешится вызов функции `print(min_val)`?

```
// ---- user.h ----
class SmallInt { /* ... */ }
void print( const SmallInt & );

// ---- user.C ----
#include <primer.h>
#include "user.h"
SmallInt asi[4];

int main() {
   // задать значения элементов массива asi

   // конкретизируется min( SmallInt*, int )
   // int size = sizeof(asi) / sizeof(SmallInt);
   min( &asi[0], size );

}
```

Это нормально: мы хотим, чтобы учитывалась именно наша функция `print(const SmallInt &)`. Рассмотрения функций, определенных в библиотеке `<primer.h>`, недостаточно. Второй шаг разрешения имени гарантирует, что если имя, использованное в определении, зависит от параметров шаблона, то принимаются во внимание имена, объявленные в контексте конкретизации. Поэтому можно быть уверенным, что функции, умеющие манипулировать объектами типа `SmallInt`, попадут в поле зрения компилятора при анализе шаблона, которому в качестве аргумента передан тип `SmallInt`.

Место в программе, где происходит конкретизация шаблона, называется *точкой конкретизации*. Знание этой точки важно потому, что она определяет, какие объявления учитывает компилятор для имен, зависящих от параметров шаблона. Такая точка всегда находится в области видимости пространства имен и следует за функцией, внутри которой произошла конкретизация. Например, точка конкретизации `min(SmallInt*,int)` расположена сразу после функции `main()` в области видимости

```
// ...
int main() {
   // ...
   // использование min(SmallInt*,int)
   min( &asi[0], size );
}
// точка конкретизации min(SmallInt*,int)
// как будто объявление конкретизированной функции выглядит так:
SmallInt min( SmallInt* array, int size )
```

пространства имен:

```
   { /* ... */ }
```

Но что, если конкретизация шаблона случается в одном исходном файле несколько раз? Где тогда будет точка конкретизации? Вы можете спросить: “А какая, собственно, разница?” В нашем примере для `SmallInt` разница есть, поскольку объявление функции `print(const SmallInt &)` должно появиться перед точкой конкретизации `min(SmallInt*,int)`:

```
#include <primer.h>
void another();

SmallInt asi[4];

int main() {
   // задать значения элементов массива asi
   int size = sizeof(asi) / sizeof(SmallInt);
   min( &asi[0], size );

   another();
   // ...
}
// точка конкретизации здесь?

void another() {
   int size = sizeof(asi) / sizeof(SmallInt);
   min( &asi[0], size );
}

// или здесь?
```

В действительности точка конкретизации находится после определения каждой функции, в которой используется конкретизированный экземпляр. Компилятор может выбрать любую из этих точек, чтобы конкретизировать в ней шаблон. Отсюда следует, что при организации кода программы надо быть внимательным и помещать все объявления, необходимые для разрешения имен, зависящих от параметров некоторого шаблона, перед первой точкой. Поэтому разумно поместить их в заголовочный файл, который

```
#include <primer.h>
// user.h содержит объявления, необходимые при конкретизации
#include "user.h"
void another();

SmallInt asi[4];

int main() {
   // ...
}
// первая точка конкретизации min(SmallInt*,int)

void another() {
   // ...
}
```

включается перед любой возможной конкретизацией шаблона:

```
// вторая точка конкретизации min(SmallInt*,int)
```

А если конкретизация шаблона происходит в нескольких файлах? Например, что будет, если функция `another()` находится в другом файле, нежели `main()`? Тогда точка конкретизации есть в каждом файле, где используется конкретизированная из шаблона функция. Компилятор свободен в выборе любой из них, так что нам снова придется проявить аккуратность и включить файл `"user.h"` во все исходные файлы, где используются конкретизированные функции. Тем самым гарантируется, что реализация `min(SmallInt*,int)` будет ссылаться именно на нашу функцию `print(const SmallInt &)` вне зависимости от того, какую из точек конкретизации выберет компилятор.

Упражнение 10.13

Назовите два шага разрешения имени в определениях шаблона. Объясните, каким образом первый шаг отвечает потребностям разработчика библиотеки, а второй обеспечивает гибкость, необходимую пользователям шаблонов.

Упражнение 10.14

На какие объявления ссылаются имена `display` и `SIZE` в реализации

```
// ---- exercise.h ----
void display( const void* );
typedef unsigned int SIZE;

template <typename Type>
   Type max( Type* array, SIZE size )
{
   Type max_val = array[0];
   for ( SIZE i = 1; i < size; ++i )
      if ( array[i] > max_val )
         max_val = array[i];

   display( "Maximum value found: " );
   display( max_val );

   return max_val;
}
// ---- user.h ----
class LongDouble { /* ... */ };
void display( const LongDouble & );
void display( const char * );
typedef int SIZE;

// ---- user.C ----
#include <exercize.h>
#include "user.h"

LongDouble ad[7];

int main() {
   // задать значения элементов массива ad

   // конкретизируется max( LongDouble*, SIZE )
   SIZE size = sizeof(ad) / sizeof(LongDouble);

   max( &ad[0], size );
}
```

`max(LongDouble*,SIZE)`?

10.10. Пространства имен и шаблоны функций A

Как и любое другое глобальное определение, шаблон функции может быть помещен в пространство имен (см. обсуждение пространств имен в разделах 8.5 и 8.6). Мы получили бы ту же семантику, если бы определили шаблон в глобальной области видимости, скрыв его имя внутри пространства имен. При использовании вне этого пространства необходимо либо квалифицировать имя шаблона именем пространства имен, либо использовать using-объявление:

```
// ---- primer.h ----
namespace cplusplus_primer {
   // определение шаблона скрыто в пространстве имен
   template <class Type>
      Type min( Type* array, int size ) { /* ... */ }
}

// ---- user.C ----
#include <primer.h>
int ai[4] = { 12, 8, 73, 45 };

int main() {
   int size = sizeof(ai) / sizeof(ai[0]);

   // ошибка: функция min() не найдена
   min( &ai[0], size );

   using cplusplus_primer::min;  // using-объявление
   // правильно: относится к min() в пространстве имен cplusplus_primer
   min( &ai[0], size );

}
```

Что произойдет, если наша программа использует шаблон, определенный в пространстве имен, и мы хотим предоставить для него специализацию? (Явные специализации шаблонов рассматривались в разделе 10.6.) Допустим, мы хотим использовать шаблон `min()`, определенный в `cplusplus_primer`, для нахождения минимального значения в массиве объектов типа `SmallInt`. Однако мы осознаем, что имеющееся определение шаблона не вполне подходит, поскольку сравнение в нем выглядит так:

```
if ( array[i] < min_val )
```

В этой инструкции два объекта класса `SmallInt` сравниваются с помощью оператора `<`. Но этот оператор неприменим к объектам, если только не перегружен в классе `SmallInt` (мы покажем, как определять перегруженные операторы в главе 15). Предположим, что мы хотели бы определить специализацию шаблона `min()`, чтобы она пользовалась

```
// функция сравнения объектов SmallInt
// возвращает true, если parm1 меньше parm2
```

функцией `compareLess()` для сравнения двух подобных объектов. Вот ее объявление:

```
bool compareLess( const SmallInt &parm1, const SmallInt &parm2 );
```

Как должно выглядеть определение этой функции? Чтобы ответить на этот вопрос, необходимо познакомиться с определением класса `SmallInt` более подробно. Данный класс позволяет определять объекты, которые хранят тот же диапазон значений, что и 8-разрядный тип `unsigned char`, т.е. от `0` до `255`. Дополнительная функциональность состоит в том, что класс перехватывает ошибки переполнения и потери значимости. Во всем остальном он должен вести себя точно так же, как `unsigned char`. Определение `SmallInt` выглядит следующим образом:

```
class SmallInt {
public:
  SmallInt( int ival ) : value( ival ) {}
  friend bool compareLess( const SmallInt &, const SmallInt & );
private:
   int value;   // член
};
```

В этом классе есть один закрытый член value, в котором хранится значение объекта типа

```
// конструктор класса SmallInt
```

SmallInt. Класс также содержит конструктор с параметром ival:

```
SmallInt( int ival ) : value( ival ) {}
```

Его единственное назначение – инициализировать член класса value значением ival.

Вот теперь можно ответить на ранее поставленный вопрос: как должна быть определена функция compareLess()? Она будет сравнивать члены value переданных ей аргументов

```
// возвращает true, если parm1 меньше parm2
bool compareLess( const SmallInt &parm1, const SmallInt &parm2 ) {
   return parm1.value < parm2.value;
```

типа SmallInt:

```
}
```

Заметим, однако, что член value является закрытым. Как может глобальная функция обратиться к закрытому члену, не нарушив инкапсуляции класса SmallInt и не вызвав тем самым ошибку компиляции? Если вы посмотрите на определение класса SmallInt, то заметите, что глобальная функция compareLess() объявлена как дружественная (friend). Если функция объявлена таким образом, то ей доступны закрытые члены класса. (Друзья классов рассматриваются в разделе 15.2.)

Теперь мы готовы определить специализацию шаблона min(). Она следующим образом

```
// специализация min() для массива объектов SmallInt
template<> SmallInt min<smallInt>( SmallInt* array, int size )
{
   SmallInt min_val = array[0];
   for (int i = 1; i < size; ++i)
      // при сравнении используется функция compareLess()
      if ( compareLess( array[i], min_val ) )
         min_val = array[i];

   print( "Minimum value found: " );
   print( min_val );

   return min_val;
```

использует функцию compareLess().

```
}
```

```
// ---- primer.h ----
namespace cplusplus_primer {
   // определение шаблона скрыто в пространстве имен
   template <class Type>
      Type min( Type* array, int size ) { /* ... */ }
}

// ---- user.h ----
class SmallInt { /* ... */ };
void print( const SmallInt & );
bool compareLess( const SmallInt &, const SmallInt & );

// ---- user.C ----
#include <primer.h>
#include "user.h"

// ошибка: это не специализация для cplusplus_primer::min()
template<> SmallInt min<smallInt>( SmallInt* array, int size )
   { /* ... */ }
```

Где мы должны объявить эту специализацию? Предположим, что здесь:

```
// ...
```

К сожалению, этот код не работает. Явная специализация шаблона функции должна быть объявлена в том пространстве имен, где определен порождающий шаблон. Поэтому мы обязаны определить специализацию `min()` в пространстве `cplusplus_primer`. В нашей программе это можно сделать двумя способами.

Напомним, что определения пространства имен не обязательно непрерывны. Мы можем

```
// ---- user.C ----
#include <primer.h>
#include "user.h"

namespace cplusplus_primer {
   // специализация для cplusplus_primer::min()
   template<> SmallInt min<smallInt>( SmallInt* array, int size )
   { /* ... */ }
}
SmallInt asi[4];

int main() {
   // задать значения элементов массива asi с помощью функции-члена set()

   using cplusplus_primer::min;    // using-объявление
   int size = sizeof(asi) / sizeof(SmallInt);
   // конкретизируется min(SmallInt*,int)
   min( &asi[0], size );
```

повторно открыть пространство имен `cplusplus_primer` для добавления специализации:

```
}
```

Можно определить специализацию так, как мы определяем любой другой член пространства имен вне определения самого пространства: квалифицировав имя члена именем объемлющего пространства.

```
// ---- user.C ----
#include <primer.h>
#include "user.h"

// специализация для cplusplus_primer::min()
// имя специализации квалифицируется
namespace {
template<> SmallInt cplusplus_primer::
                    min<smallInt>( SmallInt* array, int size )
   { /* ... */ }

// ...
```

Если вы, пользуясь библиотекой, содержащей определения шаблонов, захотите написать их специализации, то должны будете удостовериться, что их определения помещены в то же пространство имен, что и определения исходных шаблонов.

Упражнение 10.15

Поместим содержимое заголовочного файла <exercise.h> из упражнения 10.14 в пространство имен cplusplus_primer. Как надо изменить функцию main(), чтобы она могла конкретизировать шаблон max(), находящийся в cplusplus_primer?

Упражнение 10.16

Снова обращаясь к упражнению 10.14, предположим, что содержимое заголовочного файла <exercise.h> помещено в пространство имен cplusplus_primer. Допустим, мы хотим специализировать шаблон функции max() для массивов объектов класса LongDouble. Нужно, чтобы специализация шаблона использовала функцию

```
// функция сравнения объектов класса LongDouble
// возвращает true, если parm1 больше parm2
bool compareGreater( const LongDouble &parm1,
```

compareGreater() для сравнения двух объектов класса LongDouble, объявленную как:

```
                     const LongDouble &parm2 );
```

```
class LongDouble {
public:
  LongDouble(double dval) : value(ival) {}
  friend bool compareGreater( const LongDouble &,
                              const LongDouble & );
private:
   double value;
```

Определение класса LongDouble выглядит следующим образом:

```
};
```

Напишите определение функции compareGreater() и специализацию max(), в которой эта функция используется. Напишите также функцию main(), которая задает элементы массива ad, а затем вызывает специализацию max(), доставляющую его максимальный элемент. Значения, которыми инициализируется массив ad, должны быть получены чтением из стандартного ввода cin.

10.11. Пример шаблона функции

В этом разделе приводится пример, показывающий, как можно определять и использовать шаблоны функций. Здесь определяется шаблон `sort()`, который затем применяется для сортировки элементов массива. Сам массив представлен шаблоном класса `Array` (см. раздел 2.5). Таким образом, шаблоном `sort()` можно пользоваться для сортировки массивов элементов любого типа.

В главе 6 мы видели, что в стандартной библиотеке C++ определен контейнерный тип `vector`, который ведет себя во многом аналогично типу `Array`. В главе 12 рассматриваются обобщенные алгоритмы, способные манипулировать контейнерами, описанными в главе 6. Один из таких алгоритмов, `sort()`, служит для сортировки содержимого вектора. В этом разделе мы определим собственный "обобщенный алгоритм `sort()`" для манипулирования классом `Array`, упрощенной версии алгоритма из стандартной библиотеки C++.

```
template <class elemType>
   void sort( Array<elemType> &array, int low, int high ) {

   if ( low < high ) {
      int lo = low;
      int hi = high + 1;
      elemType elem = array[lo];

      for (;;) {
         while ( min( array[++lo], elem ) != elem && lo < high ) ;
         while ( min( array[--hi], elem ) == elem && hi > low ) ;

         if (lo < hi)
            swap( array, lo, hi );
         else break;
      }

      swap( array, low, hi );
      sort( array, low, hi-1 );
      sort( array, hi+1, high );
   }
```

Шаблон функции `sort()` для шаблона класса `Array` определен следующим образом:

```
}
```

В `sort()` используются две вспомогательные функции: `min()` и `swap()`. Обе они должны определяться как шаблоны, чтобы иметь возможность обрабатывать любые типы фактических аргументов, с которыми может быть конкретизирован шаблон `sort()`. `min()` определена как шаблон функции для поиска минимального из двух значений

```
template <class Type>
   Type min( Type a, Type b ) {
      return a < b ? a : b;
```

любого типа:

```
}
```

`swap()` – шаблон функции для перестановки двух элементов массива любого типа:

```
template <class elemType>
   void swap( Array<elemType> &array, int i, int j )
{
   elemType tmp = array[ i ];
   array[ i ] = array[ j ];
   array[ j ] = tmp;
}
```

Убедиться в том, что функция `sort()` действительно работает, можно с помощью отображения содержимого массива после сортировки. Поскольку функция `display()` должна обрабатывать любой массив, конкретизированный из шаблона класса `Array`, ее

```
#include <iostream>

template <class elemType>
   void display( Array<elemType> &array )
{ //формат отображения: < 0 1 2 3 4 5 >

   cout << "< ";
   for ( int ix = 0; ix < array.size(); ++ix )
      cout << array[ix] << " ";
   cout << ">\n";
}
```

тоже следует определить как шаблон:

В этом примере мы пользуемся моделью компиляции с включением и помещаем шаблоны всех функций в заголовочный файл `Array.h` вслед за объявлением шаблона класса `Array`.

Следующий шаг – написание функции для тестирования этих шаблонов. В `sort()` поочередно передаются массивы элементов типа `double`, типа `int` и массив строк. Вот текст программы:

```
#include <iostream>
#include <string>
#include "Array.h"

double da[10] = {
   26.7, 5.7, 37.7, 1.7, 61.7, 11.7, 59.7,
   15.7, 48.7, 19.7 };

int ia[16] = {
   503, 87, 512, 61, 908, 170, 897, 275, 653,
   426, 154, 509, 612, 677, 765, 703 };

string sa[11] = {
   "a", "heavy", "snow", "was", "falling", "when",
   "they", "left", "the", "police", "station" };

int main() {

   // вызвать конструктор для инициализации arrd
   Array<double> arrd( da, sizeof(da)/sizeof(da[0]) );

   // вызвать конструктор для инициализации arri
   Array<int> arri( ia, sizeof(ia)/sizeof(ia[0]) );

   // вызвать конструктор для инициализации arrs
   Array<string> arrs( sa, sizeof(sa)/sizeof(sa[0]) );

   cout << "sort array of doubles (size == "
        << arrd.size() << ")" << endl;
   sort(arrd, 0, arrd.size()-1 );
   display(arrd);

   cout << "sort array of ints (size == "
        << arri.size() << ")" << endl;
   sort(arri, 0, arri.size()-1 );
   display(arri);

   cout << "sort array of strings (size == "
        << arrs.size() << ")" << endl;
   sort(arrs, 0, arrs.size()-1 );
   display(arrs);

   return 0;
}
```

Если скомпилировать и запустить программу, то она напечатает следующее (эти строки искусственно разбиты на небольшие части):

```
sort array of doubles (size == 10)
< 1.7 5.7 11.7 14.9 15.7 19.7 26.7
  37.7 48.7 59.7 61.7 >

sort array of ints (size == 16)
< 61 87 154 170 275 426 503 509 512
  612 653 677 703 765 897 908 >

sort array of strings (size == 11)
< "a" "falling" "heavy" "left" "police" "snow"
  "station" "the" "they" "was" "when" >
```

В числе обобщенных алгоритмов, имеющихся в стандартной библиотеке C++ (и в главе 12), вы найдете также функции `min()` и `swap()`. В главе 12 мы покажем, как их использовать.

11

11. Обработка исключений

Обработка исключений – это механизм, позволяющий двум независимо разработанным программным компонентам взаимодействовать в аномальной ситуации, называемой *исключением*. В этой главе мы расскажем, как генерировать, или возбуждать, исключение в том месте программы, где имеет место аномалия. Затем мы покажем, как связать `catch`-обработчик исключений с множеством инструкций программы, используя `try`-блок. Потом речь пойдет о спецификации исключений – механизме, с помощью которого можно связать список исключений с объявлением функции, и функция не сможет возбудить никаких других исключений. Закончится эта глава обсуждением решений, принимаемых при проектировании программы, в которой используются исключения.

11.1. Возбуждение исключения

Исключение – это аномальное поведение во время выполнения, которое программа может обнаружить, например: деление на 0, выход за границы массива или истощение свободной памяти. Такие исключения нарушают нормальный ход работы программы, и на них нужно немедленно отреагировать. В C++ имеются встроенные средства для их возбуждения и обработки. С помощью этих средств активизируется механизм, позволяющий двум несвязанным (или независимо разработанным) фрагментам программы обмениваться информацией об исключении.

Когда встречается аномальная ситуация, та часть программы, которая ее обнаружила, может сгенерировать, или *возбудить*, исключение. Чтобы понять, как это происходит, реализуем по-новому класс `iStack`, представленный в разделе 4.15, используя исключения для извещения об ошибках при работе со стеком. Определение класса `iStack`

```
#include <vector>
class iStack {
public:
   iStack( int capacity )
         : _stack( capacity ), _top( 0 ) { }

   bool pop( int &top_value );
   bool push( int value );

   bool full();
   bool empty();
   void display();

   int size();

private:
   int _top;
   vector< int > _stack;
```

выглядит следующим образом:

```
};
```

Стек реализован на основе вектора из элементов типа int. При создании объекта класса iStack его конструктор создает вектор из int, размер которого (максимальное число элементов, хранящихся в стеке) задается с помощью начального значения. Например, следующая инструкция создает объект myStack, который способен содержать не более 20 элементов типа int:

```
iStack myStack(20);
```

При манипуляциях с объектом myStack могут возникнуть две ошибки:

- запрашивается операция pop(), но стек пуст;
- запрашивается операция push(), но стек полон.

Вызвавшую функцию нужно уведомить об этих ошибках посредством исключений. С чего же начать?

Во-первых, мы должны определить, какие именно исключения могут быть возбуждены. В C++ они чаще всего реализуются с помощью классов. Хотя в полном объеме классы будут представлены в главе 13, мы все же определим здесь два из них, чтобы использовать их как исключения для класса iStack. Эти определения мы поместим в

```
// stackExcp.h
class popOnEmpty { /* ... */ };
```

заголовочный файл stackExcp.h:

```
class pushOnFull { /* ... */ };
```

В главе 19 исключения в виде классов обсуждаются более подробно, там же рассматривается иерархия таких классов, предоставляемая стандартной библиотекой C++.

Затем надо изменить определения функций-членов pop() и push() так, чтобы они возбуждали эти исключения. Для этого предназначена инструкция throw, которая во многих отношениях напоминает return. Она состоит из ключевого слова throw, за которым следует выражение того же типа, что и тип возбуждаемого исключения. Как

```
// увы, это не совсем правильно
```

выглядит инструкция throw для функции pop()? Попробуем такой вариант:

```
throw popOnEmpty;
```

К сожалению, так нельзя. Исключение – это объект, и функция pop() должна генерировать объект класса соответствующего типа. Выражение в инструкции throw не может быть просто типом. Для создания нужного объекта необходимо вызвать

```
// инструкция является вызовом конструктора
```

конструктор класса. Инструкция throw для функции pop() будет выглядеть так:

```
throw popOnEmpty();
```

Эта инструкция создает объект исключения типа popOnEmpty.

Напомним, что функции-члены pop() и push() были определены как возвращающие значение типа bool: true означало, что операция завершилась успешно, а false – что произошла ошибка. Поскольку теперь для извещения о неудаче pop() и push() используют исключения, возвращать значение необязательно. Поэтому мы будем считать,

```
class iStack {
public:
   // ...

   // больше не возвращают значения
   void pop( int &value );
   void push( int value );

private:
   // ...
```

что эти функции-члены имеют тип void:

```
};
```

Теперь функции, пользующиеся нашим классом iStack, будут предполагать, что все хорошо, если только не возбуждено исключение; им больше не надо проверять возвращенное значение, чтобы узнать, как завершилась операция. В двух следующих разделах мы покажем, как определить функцию для обработки исключений, а сейчас

```
#include "stackExcp.h"

void iStack::pop( int &top_value )
{
   if ( empty() )
      throw popOnEmpty();

   top_value = _stack[ --_top ];

   cout << "iStack::pop(): " << top_value << endl;
}

void iStack::push( int value )
{
   cout << "iStack::push( " << value << " )\n";

   if ( full() )
      throw pushOnFull( value );

   _stack[ _top++ ] = value;
```

представим новые реализации функций-членов pop() и push() класса iStack:

```
}
```

Хотя исключения чаще всего представляют собой объекты типа класса, инструкция throw может генерировать объекты любого типа. Например, функция mathFunc() в следующем примере возбуждает исключение в виде объекта-перечисления . Это корректный код C++:

```
enum EHstate { noErr, zeroOp, negativeOp, severeError };

int mathFunc( int i ) {
   if ( i == 0 )
      throw zeroOp;  // исключение в виде объекта-перечисления

   // в противном случае продолжается нормальная обработка

}
```

Упражнение 11.1

Какие из приведенных инструкций `throw` ошибочны? Почему? Для правильных

```
(a) class exceptionType { };
    throw exceptionType();
(b) int excpObj;
    throw excpObj;
(c) enum mathErr { overflow, underflow, zeroDivide };
    throw mathErr zeroDivide();
(d) int *pi = excpObj;
```

инструкций укажите тип возбужденного исключения:

```
    throw pi;
```

Упражнение 11.2

У класса `IntArray`, определенного в разделе 2.3, имеется функция-оператор `operator[]()`, в которой используется `assert()` для извещения о том, что индекс вышел за пределы массива. Измените определение этого оператора так, чтобы в подобной ситуации он генерировал исключение. Определите класс, который будет употребляться как тип возбужденного исключения.

11.2. try-блок

В нашей программе тестируется определенный в предыдущем разделе класс `iStack` и его функции-члены `pop()` и `push()`. Выполняется 50 итераций цикла `for`. На каждой итерации в стек помещается значение, кратное 3: 3, 6, 9 и т.д. Если значение кратно 4 (4, 8, 12...), то выводится текущее содержимое стека, а если кратно 10 (10, 20, 30...), то с вершины снимается один элемент, после чего содержимое стека выводится снова. Как нужно изменить функцию `main()`, чтобы она обрабатывала исключения, возбуждаемые функциями-членами класса `iStack`?

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

int main() {
   iStack stack( 32 );

   stack.display();
   for ( int ix = 1; ix < 51; ++ix )
   {
            if ( ix % 3 == 0 )
          stack.push( ix );

            if ( ix % 4 == 0 )
               stack.display();

            if ( ix % 10  == 0 ) {
               int dummy;
               stack.pop( dummy );
               stack.display();
            }
   }
   return 0;
}
```

Инструкции, которые могут возбуждать исключения, должны быть заключены в *try-блок*. Такой блок начинается с ключевого слова `try`, за которым идет последовательность инструкций, заключенная в фигурные скобки, а после этого – список обработчиков, называемых *catch-предложениями*. Try-блок группирует инструкции программы и ассоциирует с ними обработчики исключений. Куда нужно поместить try-блоки в

```
for ( int ix = 1; ix < 51; ++ix ) {
   try {   // try-блок для исключений pushOnFull
           if ( ix % 3 == 0 )
         stack.push( ix );
   }
   catch ( pusOnFull ) { ... }

   if ( ix % 4 == 0 )
      stack.display();

   try {   // try-блок для исключений popOnEmpty
           if ( ix % 10  == 0 ) {
              int dummy;
              stack.pop( dummy );
              stack.display();
           }
   }
   catch ( popOnEmpty ) { ... }
```

функции `main()`, чтобы были обработаны исключения `popOnEmpty` и `pushOnFull`?

```
}
```

В таком виде программа выполняется корректно. Однако обработка исключений в ней перемежается с кодом, использующимся при нормальных обстоятельствах, а такая организация несовершенна. В конце концов, исключения – это аномальные ситуации, возникающие только в особых случаях. Желательно отделить код для обработки

аномалий от кода, реализующего операции со стеком. Мы полагаем, что показанная ниже

```
try {
   for ( int ix = 1; ix < 51; ++ix )
   {
      if ( ix % 3 == 0 )
         stack.push( ix );

      if ( ix % 4 == 0 )
               stack.display();

      if ( ix % 10  == 0 ) {
               int dummy;
               stack.pop( dummy );
               stack.display();
      }
   }
}
catch ( pushOnFull ) { ... }
```

схема облегчает чтение и сопровождение программы:

```
catch ( popOnEmpty ) { ... }
```

С try-блоком ассоциированы два catch-предложения, которые могут обработать исключения `pushOnFull` и `popOnEmpty`, возбуждаемые функциями-членами `push()` и `pop()` внутри этого блока. Каждый catch-обработчик определяет тип "своего" исключения. Код для обработки исключения помещается внутрь составной инструкции (между фигурными скобками), которая является частью catch-обработчика. (Подробнее catch-предложения мы рассмотрим в следующем разделе.)

Исполнение программы может пойти по одному из следующих путей:

- если исключение не возбуждено, то выполняется код внутри try-блока, а ассоциированные с ним обработчики игнорируются. Функция `main()` возвращает `0`;
- если функция-член `push()`, вызванная из первой инструкции `if` внутри цикла `for`, возбуждает исключение, то вторая и третья инструкции `if` игнорируются, управление покидает цикл `for` и try-блок, и выполняется обработчик исключений типа `pushOnFull`;
- если функция-член `pop()`, вызванная из третьей инструкции `if` внутри цикла `for`, возбуждает исключение, то вызов `display()` игнорируется, управление покидает цикл `for` и try-блок, и выполняется обработчик исключений типа `popOnEmpty`.

Когда возбуждается исключение, пропускаются все инструкции, следующие за той, где оно было возбуждено. Исполнение программы возобновляется в catch-обработчике этого исключения. Если такого обработчика не существует, то управление передается в функцию `terminate()`, определенную в стандартной библиотеке C++.

Try-блок может содержать любую инструкцию языка C++: как выражения, так и объявления. Он вводит локальную область видимости, так что объявленные внутри него переменные недоступны вне этого блока, в том числе и в catch-обработчиках. Например, функцию `main()` можно переписать так, что объявление переменной `stack` окажется в try-блоке. В таком случае обращаться к этой переменной в catch-обработчиках нельзя:

```
int main() {
   try {
      iStack stack( 32 );    // правильно: объявление внутри try-блока

      stack.display();
      for ( int ix = 1; ix < 51; ++ix )
      {
          // то же, что и раньше
      }
   }
   catch ( pushOnFull ) {
      // здесь к переменной stack обращаться нельзя
   }
   catch ( popOnEmpty ) {
      // здесь к переменной stack обращаться нельзя
   }

   // и здесь к переменной stack обращаться нельзя
   return 0;
}
```

Можно объявить функцию так, что все ее тело будет заключено в try-блок. При этом не обязательно помещать try-блок внутрь определения функции, удобнее заключить ее тело в *функциональный try-блок*. Такая организация поддерживает наиболее чистое разделение

```
int main()
   try {
      iStack stack( 32 );    // правильно: объявление внутри try-блока

      stack.display();
      for ( int ix = 1; ix < 51; ++ix )
      {
          // то же, что и раньше
      }

      return 0;
   }
   catch ( pushOnFull ) {
      // здесь к переменной stack обращаться нельзя
   }
   catch ( popOnEmpty ) {
      // здесь к переменной stack обращаться нельзя
   }
```

кода для нормальной обработки и кода для обработки исключений. Например:

Обратите внимание, что ключевое слово `try` находится перед фигурной скобкой, открывающей тело функции, а catch-обработчики перечислены после закрывающей его скобки. Как видим, код, осуществляющий нормальную обработку, находится внутри тела функции и четко отделен от кода для обработки исключений. Однако к переменным, объявленным в `main()`, нельзя обратиться из обработчиков исключений.

Функциональный try-блок ассоциирует группу catch-обработчиков с телом функции. Если инструкция возбуждает исключение, то поиск обработчика, способного перехватить это исключение, ведется среди тех, что идут за телом функции. Функциональные try-блоки особенно полезны в сочетании с конструкторами классов. (Мы еще вернемся к этой теме в главе 19.)

Упражнение 11.3

Напишите программу, которая определяет объект `IntArray` (тип класса `IntArray` рассматривался в разделе 2.3) и выполняет описанные ниже действия.

Пусть есть три файла, содержащие целые числа.

1. Прочитать первый файл и поместить в объект `IntArray` первое, третье, пятое, ..., n-ое значение (где n нечетно). Затем вывести содержимое объекта `IntArray`.
2. Прочитать второй файл и поместить в объект `IntArray` пятое, десятое, ..., n-ое значение (где n кратно 5). Вывести содержимое объекта.
3. Прочитать третий файл и поместить в объект `IntArray` второе, четвертое, ..., n-ое значение (где n четно). Вывести содержимое объекта.

Воспользуйтесь оператором `operator[]()` класса `IntArray`, определенным в упражнении 11.2, для сохранения и получения значений из объекта `IntArray`. Так как `operator[]()` может возбуждать исключения, обработайте их, поместив необходимое количество try-блоков и catch-обработчиков. Объясните, почему вы разместили try-блоки именно так, а не иначе.

11.3. Перехват исключений

В языке C++ исключения обрабатываются в предложениях *catch*. Когда какая-то инструкция внутри try-блока возбуждает исключение, то просматривается список последующих предложений `catch` в поисках такого, который может его обработать.

Catch-обработчик состоит из трех частей: ключевого слова `catch`, объявления одного типа или одного объекта, заключенного в круглые скобки (оно называется *объявлением исключения*), и составной инструкции. Если для обработки исключения выбрано некоторое catch-предложение, то выполняется эта составная инструкция. Рассмотрим catch-обработчики исключений `pushOnFull` и `popOnEmpty` в функции `main()` более

```
catch ( pushOnFull ) {
   cerr << "trying to push value on a full stack\n";
   return errorCode88;
}
catch ( popOnEmpty ) {
   cerr << "trying to pop a value on an empty stack\n";
   return errorCode89;
```

подробно:

```
}
```

В обоих catch-обработчиках есть объявление типа класса; в первом это `pushOnFull`, а во втором – `popOnEmpty`. Для обработки исключения выбирается тот обработчик, для которого типы в объявлении исключения и в возбужденном исключении совпадают. (В главе 19 мы увидим, что типы не обязаны совпадать точно: обработчик для базового класса подходит и для исключений с производными классами.) Например, когда функция-член `pop()` класса `iStack` возбуждает исключение `popOnEmpty`, то управление попадает во второй обработчик. После вывода сообщения об ошибке в `cerr`, функция `main()` возвращает код `errorCode89`.

А если catch-обработчики не содержат инструкции `return`, с какого места будет продолжено выполнение программы? После завершения обработчика выполнение

возобновляется с инструкции, идущей за последним catch-обработчиком в списке. В нашем примере оно продолжается с инструкции return в функции main(). После того

```
int main() {
   iStack stack( 32 );

   try {
      stack.display();
      for ( int x = 1; ix < 51; ++ix )
      {
         // то же, что и раньше
      }
   }
   catch ( pushOnFull ) {
      cerr << "trying to push value on a full stack\n";
   }
   catch ( popOnEmpty ) {
      cerr << "trying to pop a value on an empty stack\n";
   }

   // исполнение продолжается отсюда
   return 0;
}
```

как catch-обработчик popOnEmpty выведет сообщение об ошибке, main() вернет 0.

Говорят, что механизм обработки исключений в C++ *невозвратный*: после того как исключение обработано, управление не возобновляется с того места, где оно было возбуждено. В нашем примере управление не возвращается в функцию-член pop(), возбудившую исключение.

11.3.1. Объекты-исключения

Объявлением исключения в catch-обработчике могут быть объявления типа или объекта. В каких случаях это следует делать? Тогда, когда необходимо получить значение или как-то манипулировать объектом, созданным в выражении throw. Если классы исключений спроектированы так, что в объектах-исключениях при возбуждении сохраняется некоторая информация и если в объявлении исключения фигурирует такой объект, то инструкции внутри catch-обработчика могут обращаться к информации, сохраненной в объекте выражением throw.

Изменим реализацию класса исключения pushOnFull, сохранив в объекте-исключении то значение, которое не удалось поместить в стек. Catch-обработчик, сообщая об ошибке, теперь будет выводить его в cerr. Для этого мы сначала модифицируем определение

```
// новый класс исключения:
// он сохраняет значение, которое не удалось поместить в стек
class pushOnFull {
public:
   pushOnFull( int i ) : _value( i ) { }
   int value { return _value; }
private:
   int _value;
```

типа класса pushOnFull следующим образом:

```
};
```

Новый закрытый член _value содержит число, которое не удалось поместить в стек. Конструктор принимает значение типа int и сохраняет его в члене _data. Вот как

```
void iStack::push( int value )
{
   if ( full() )
      // значение, сохраняемое в объекте-исключении
      throw pushOnFull( value );

   // ...
```

вызывается этот конструктор для сохранения значения из выражения throw:

```
}
```

У класса pushOnFull появилась также новая функция-член value(), которую можно использовать в catch-обработчике для вывода хранящегося в объекте-исключении

```
catch ( pushOnFull eObj ) {
   cerr << "trying to push value " << eObj.value()
        << " on a full stack\n";
```

значения:

```
}
```

Обратите внимание, что в объявлении исключения в catch-обработчике фигурирует объект eObj, с помощью которого вызывается функция-член value() класса pushOnFull.

Объект-исключение всегда создается в точке возбуждения, даже если выражение throw –

```
enum EHstate { noErr, zeroOp, negativeOp, severeError };
enum EHstate state = noErr;

int mathFunc( int i ) {
   if ( i == 0 ) {
      state = zeroOp;
      throw state;    // создан объект-исключение
   }
   // иначе продолжается обычная обработка
```

это не вызов конструктора и, на первый взгляд, не должно создавать объекта. Например:

```
}
```

В этом примере объект state не используется в качестве объекта-исключения. Вместо этого выражением throw создается объект-исключение типа EHstate, который инициализируется значением глобального объекта state. Как программа может различить их? Для ответа на этот вопрос мы должны присмотреться к объявлению исключения в catch-обработчике более внимательно.

Это объявление ведет себя почти так же, как объявление формального параметра. Если при входе в catch-обработчик исключения выясняется, что в нем объявлен объект, то он инициализируется копией объекта-исключения. Например, следующая функция

calculate() вызывает определенную выше mathFunc(). При входе в catch-обработчик внутри calculate() объект eObj инициализируется копией объекта-исключения,

```
void calculate( int op ) {
   try {
      mathFunc( op );
   }
   catch ( EHstate eObj ) {
      // eObj - копия сгенерированного объекта-исключения
   }
```

созданного выражением throw.

```
}
```

Объявление исключения в этом примере напоминает передачу параметра по значению. Объект eObj инициализируется значением объекта-исключения точно так же, как переданный по значению формальный параметр функции – значением соответствующего фактического аргумента. (Передача параметров по значению рассматривалась в разделе 7.3.)

Как и в случае параметров функции, в объявлении исключения может фигурировать ссылка. Тогда catch-обработчик будет напрямую ссылаться на объект-исключение,

```
void calculate( int op ) {
try {
      mathFunc( op );
   }
   catch ( EHstate &eObj ) {
      // eObj ссылается на сгенерированный объект-исключение
   }
```

сгенерированный выражением throw, а не создавать его локальную копию:

```
}
```

Для предотвращения ненужного копирования больших объектов применять ссылки следует не только в объявлениях параметров типа класса, но и в объявлениях исключений того же типа.

В последнем случае catch-обработчик сможет модифицировать объект-исключение. Однако переменные, определенные в выражении throw, остаются без изменения. Например, модификация eObj внутри catch-обработчика не затрагивает глобальную

```
void calculate( int op ) {
try {
      mathFunc( op );
   }
   catch ( EHstate &eObj ) {
      // исправить ошибку, вызвавшую исключение
      eObj = noErr;  // глобальная переменная state не изменилась
   }
```

переменную state, установленную в выражении throw:

```
}
```

Catch-обработчик переустанавливает eObj в noErr после исправления ошибки, вызвавшей исключение. Поскольку eObj – это ссылка, можно ожидать, что присваивание модифицирует глобальную переменную state. Однако изменяется лишь объект-исключение, созданный в выражении throw, поэтому модификация eObj не затрагивает state.

11.3.2. Раскрутка стека

Поиск catch-обработчикадля возбужденного исключения происходит следующим образом. Когда выражение throw находится в try-блоке, все ассоциированные с ним предложения catch исследуются с точки зрения того, могут ли они обработать исключение. Если подходящее предложение catch найдено, то исключение обрабатывается. В противном случае поиск продолжается в вызывающей функции. Предположим, что вызов функции, выполнение которой прекратилось в результате исключения, погружен в try-блок; в такой ситуации исследуются все предложения catch, ассоциированные с этим блоком. Если один из них может обработать исключение, то процесс заканчивается. В противном случае переходим к следующей по порядку вызывающей функции. Этот поиск последовательно проводится во всей цепочке вложенных вызовов. Как только будет найдено подходящее предложение, управление передается в соответствующий обработчик.

В нашем примере первая функция, для которой нужен catch-обработчик, – это функция-член pop() класса iStack. Поскольку выражение throw внутри pop() не находится в try-блоке, то программа покидает pop(), не обработав исключение. Следующей рассматривается функция, вызвавшая pop(), то есть main(). Вызов pop() внутри main() находится в try-блоке, и далее исследуется, может ли хотя бы одно ассоциированное с ним предложение catch обработать исключение. Поскольку обработчик исключения popOnEmpty имеется, то управление попадает в него.

Процесс, в результате которого программа последовательно покидает составные инструкции и определения функций в поисках предложения catch, способного обработать возникшее исключение, называется *раскруткой стека*. По мере раскрутки прекращают существование локальные объекты, объявленные в составных инструкциях и определениях функций, из которых произошел выход. C++ гарантирует, что во время описанного процесса вызываются деструкторы локальных объектов классов, хотя они исчезают из-за возбужденного исключения. (Подробнее мы поговорим об этом в главе 19.)

Если в программе нет предложения catch, способного обработать исключение, оно остается необработанным. Но исключение – это настолько серьезная ошибка, что программа не может продолжать выполнение. Поэтому, если обработчик не найден, вызывается функция terminate() из стандартной библиотеки C++. По умолчанию terminate() активизирует функцию abort(), которая аномально завершает программу. (В большинстве ситуаций вызов abort() оказывается вполне приемлемым решением. Однако иногда необходимо переопределить действия, выполняемые функцией terminate(). Как это сделать, рассказывается в книге [STROUSTRUP97].)

Вы уже, наверное, заметили, что обработка исключений и вызов функции во многом похожи. Выражение throw ведет себя аналогично вызову, а предложение catch чем-то напоминает определение функции. Основная разница между этими двумя механизмами заключается в том, что информация, необходимая для вызова функции, доступна во время компиляции, а для обработки исключений – нет. Обработка исключений в C++ требует языковой поддержки во время выполнения. Например, для обычного вызова

функции компилятору в точке активизации уже известно, какая из перегруженных функций будет вызвана. При обработке же исключения компилятор не знает, в какой функции находится catch-обработчик и откуда возобновится выполнение программы. Функция `terminate()` предоставляет механизм времени выполнения, который извещает пользователя о том, что подходящего обработчика не нашлось.

11.3.3. Повторное возбуждение исключения

Может оказаться так, что в одном предложении `catch` не удалось полностью обработать исключение. Выполнив некоторые корректирующие действия, catch-обработчик может решить, что дальнейшую обработку следует поручить функции, расположенной "выше" в цепочке вызовов. Передать исключение другому catch-обработчику можно с помощью *повторного возбуждения исключения*. Для этой цели в языке предусмотрена конструкция

```
throw;
```

которая вновь генерирует объект-исключение. Повторное возбуждение возможно только

```
catch ( exception eObj ) {
   if ( canHandle( eObj ) )
      // обработать исключение
      return;
   else
      // повторно возбудить исключение, чтобы его перехватил другой
      // catch-обработчик
      throw;
}
```

внутри составной инструкции, являющейся частью catch-обработчика:

При повторном возбуждении новый объект-исключение не создается. Это имеет значение, если catch-обработчик модифицирует объект, прежде чем возбудить исключение

```
enum EHstate { noErr, zeroOp, negativeOp, severeError };

void calculate( int op ) {
try {
      // исключение, возбужденное mathFunc(), имеет значение zeroOp
      mathFunc( op );
   }
   catch ( EHstate eObj ) {
      // что-то исправить

      // пытаемся модифицировать объект-исключение
      eObj = severeErr;

      // предполагалось, что повторно возбужденное исключение будет
      // иметь значение severeErr
      throw;
   }
```

повторно. В следующем фрагменте исходный объект-исключение не изменяется. Почему?

```
}
```

Так как eObj не является ссылкой, то catch-обработчик получает копию объекта-исключения, так что любые модификации eObj относятся к локальной копии и не отражаются на исходном объекте-исключении, передаваемом при повторном возбуждении. Таким образом, переданный далее объект по-прежнему имеет тип zeroOp.

Чтобы модифицировать исходный объект-исключение, в объявлении исключения внутри

```
catch ( EHstate &eObj ) {
   // модифицируем объект-исключение
   eObj = severeErr;

   // повторно возбужденное исключение имеет значение severeErr
   throw;
```

catch-обработчика должна фигурировать ссылка:

```
}
```

Теперь eObj ссылается на объект-исключение, созданный выражением throw, так что все изменения относятся непосредственно к исходному объекту. Поэтому при повторном возбуждении исключения далее передается модифицированный объект.

Таким образом, другая причина для объявления ссылки в catch-обработчике заключается в том, что сделанные внутри обработчика модификации объекта-исключения в таком случае будут видны при повторном возбуждении исключения. (Третья причина будет рассмотрена в разделе 19.2, где мы расскажем, как catch-обработчик вызывает виртуальные функции класса.)

11.3.4. Перехват всех исключений

Иногда функции нужно выполнить определенное действие до того, как она завершит обработку исключения, даже несмотря на то, что обработать его она не может. К примеру, функция захватила некоторый ресурс, скажем открыла файл или выделила

```
void manip() {
   resource res;
   res.lock();       // захват ресурса

   // использование ресурса
   // действие, в результате которого возбуждено исключение

   res.release();    // не выполняется, если возбуждено исключение
```

память из хипа, и этот ресурс необходимо освободить перед выходом:

```
}
```

Если исключение возбуждено, то управление не попадет на инструкцию, где ресурс освобождается. Чтобы освободить ресурс, не пытаясь перехватить все возможные исключения (тем более, что мы не всегда знаем, какие именно исключения могут возникнуть), воспользуемся специальной конструкцией, позволяющей перехватывать любые исключения. Это не что иное, как предложение catch, в котором объявление исключения имеет вид (...) и куда управление попадает при любом исключении. Например:

```
// управление попадает сюда при любом возбужденном исключении
catch (...) {
   // здесь размещаем наш код
}
```

Конструкция `catch(...)` используется в сочетании с повторным возбуждением исключения. Захваченный ресурс освобождается внутри составной инструкции в catch-обработчике перед тем, как передать исключение по цепочке вложенных вызовов в

```
void manip() {
   resource res;
   res.lock();
   try {
      // использование ресурса
      // действие, в результате которого возбуждено исключение
   }
   catch (...) {
      res.release();
      throw;
   }
   res.release();   // не выполняется, если возбуждено исключение
}
```

результате повторного возбуждения:

Чтобы гарантировать освобождение ресурса в случае, когда выход из `manip()` происходит в результате исключения, мы освобождаем его внутри `catch(...)` до того, как исключение будет передано дальше. Можно также управлять захватом и освобождением ресурса путем инкапсуляции в класс всей работы с ним. Тогда захват будет реализован в конструкторе, а освобождение – в автоматически вызываемом деструкторе. (С этим подходом мы познакомимся в главе 19.)

Предложение `catch(...)` используется самостоятельно или в сочетании с другими catch-обработчиками. В последнем случае следует позаботиться о правильной организации обработчиков, ассоциированных с try-блоком.

Catch-обработчики исследуются по очереди, в том порядке, в котором они записаны. Как только найден подходящий, просмотр прекращается. Следовательно, если предложение `catch(...)` употребляется вместе с другими catch-обработчиками, то оно должно быть

```
try {
   stack.display();
   for ( int ix = 1; ix < 51; ++x )
   {
      // то же, что и выше
   }
}
catch ( pushOnFull ) { }
catch ( popOnEmpty ) { }
```

последним в списке, иначе компилятор выдаст сообщение об ошибке:

```
catch ( ... ) { }   // должно быть последним в списке catch-обработчиков
```

Упражнение 11.4

Объясните, почему модель обработки исключений в C++ называется невозвратной.

Упражнение 11.5

Даны следующие объявления исключений. Напишите выражения throw, создающие

```
(a) class exceptionType { };
    catch( exceptionType *pet ) { }
(b) catch(...) { }
(c) enum mathErr { overflow, underflow, zeroDivide };
    catch( mathErr &ref ) { }
(d) typedef int EXCPTYPE;
```

объект-исключение, который может быть перехвачен указанными обработчиками:

```
    catch( EXCPTYPE ) { }
```

Упражнение 11.6

Объясните, что происходит во время раскрутки стека.

Упражнение 11.7

Назовите две причины, по которым объявление исключения в предложении catch следует делать ссылкой.

Упражнение 11.8

На основе кода, написанного вами в упражнении 11.3, модифицируйте класс созданного исключения: неправильный индекс, использованный в операторе operator[](), должен сохраняться в объекте-исключении и затем выводиться catch-обработчиком. Измените программу так, чтобы operator[]() возбуждал при ее выполнении исключение.

11.4. Спецификации исключений

По объявлениям функций-членов pop() и push() класса iStack невозможно определить, что они возбуждают исключения. Можно, конечно, включить в объявление подходящий комментарий. Тогда описание интерфейса класса в заголовочном файле будет содержать

```
class iStack {
public:
   // ...

   void pop( int &value );   // возбуждает popOnEmpty
   void push( int value );   // возбуждает pushOnFull

private:
   // ...
```

документацию возбуждаемых исключений:

```
};
```

Но такое решение несовершенно. Неизвестно, будет ли обновлена документация при выпуске следующих версий iStack. Кроме того, комментарий не дает компилятору достоверной информации о том, что никаких других исключений функция не возбуждает. *Спецификация исключений* позволяет перечислить в объявлении функции все

исключения, которые она может возбуждать. При этом гарантируется, что другие исключения функция возбуждать не будет.

Такая спецификация следует за списком формальных параметров функции. Она состоит из ключевого слова throw, за которым идет список типов исключений, заключенный в скобки. Например, объявления функций-членов класса iStack можно модифицировать,

```
class iStack {
public:
   // ...

   void pop( int &value ) throw(popOnEmpty);
   void push( int value ) throw(pushOnFull);

private:
   // ...
```

добавив спецификации исключений:

```
};
```

Гарантируется, что при обращении к pop() не будет возбуждено никаких исключений, кроме popOnEmpty, а при обращении к push()–только pushOnFull.

Объявление исключения – это часть интерфейса функции, оно должно быть задано при ее объявлении в заголовочном файле. Спецификация исключений – это своего рода "контракт" между функцией и остальной частью программы, гарантия того, что функция не будет возбуждать никаких исключений, кроме перечисленных.

Если в объявлении функции присутствует спецификация исключений, то при повторном объявлении этой же функции должны быть перечислены точно те же типы. Спецификации исключений в разных объявлениях одной и той же функции не

```
// два объявления одной и той же функции
extern int foo( int = 0 ) throw(string);

// ошибка: опущена спецификация исключений
```

суммируются:

```
extern int foo( int parm ) { }
```

Что произойдет, если функция возбудит исключение, не перечисленное в ее спецификации? Исключения возбуждаются только при обнаружении определенных аномалий в поведении программы, и во время компиляции неизвестно, встретится ли то или иное исключение во время выполнения. Поэтому нарушения спецификации исключений функции могут быть обнаружены только во время выполнения. Если функция возбуждает исключение, не указанное в спецификации, то вызывается unexpected() из стандартной библиотеки C++, а та по умолчанию вызывает terminate(). (В некоторых случаях необходимо переопределить действия, выполняемые функцией unexpected(). Стандартная библиотека предоставляет механизм для этого. Подробнее см. [STRAUSTRUP97].)

Необходимо уточнить, что unexpected() не вызывается только потому, что функция возбудила исключение, не указанное в ее спецификации. Все нормально, если она обработает это исключение самостоятельно, внутри функции. Например:

```
void recoup( int op1, int op2 ) throw(ExceptionType)
{
   try {
      // ...
      throw string("we're in control");
   }
   // обрабатывается возбужденное исключение
   catch ( string ) {
      // сделать все необходимое
   }
}  // все хорошо, unexpected() не вызывается
```

Функция `recoup()` возбуждает исключение типа `string`, несмотря на его отсутствие в спецификации. Поскольку это исключение обработано в теле функции, `unexpected()` не вызывается.

Нарушения спецификации исключений функции обнаруживаются только во время выполнения. Компилятор не сообщает об ошибке, если в выражении `throw` возбуждается исключение неуказанного типа. Если такое выражение никогда не выполнится или не возбудит исключения, нарушающего спецификацию, то программа будет работать, как и

```
extern void doit( int, int ) throw(string, exceptionType);

void action ( int op1, int op2 ) throw(string) {
   doit( op1, op2 );   // ошибки компиляции не будет
   // ...
}
```

ожидалось, и нарушение никак не проявится:

`doit()` может возбудить исключение типа `exceptionType`, которое не разрешено спецификацией `action()`. Однако функция компилируется успешно. Компилятор при этом генерирует код, гарантирующий, что при возбуждении исключения, нарушающего спецификацию, будет вызвана библиотечная функция `unexpected()`.

Пустая спецификация показывает, что функция не возбуждает никаких исключений:

```
extern void no_problem () throw();
```

Если же в объявлении функции спецификация исключений отсутствует, то может быть возбуждено исключение любого типа.

Между типом возбужденного исключения и типом исключения, указанного в

```
int convert( int parm ) throw(string)
{
   //...
   if ( somethingRather )
      // ошибка программы:
      // convert() не допускает исключения типа const char*
      throw "help!";
}
```

спецификации, не разрешается проводить никаких преобразований:

Выражение throw в функции convert() возбуждает исключение типа строки символов в стиле языка C. Созданный объект-исключение имеет тип const char*. Обычно выражение типа const char* можно привести к типу string. Однако спецификация не допускает преобразования типов, поэтому если convert() возбуждает такое исключение, то вызывается unexpected(). Для исправления ошибки выражение throw можно модифицировать так, чтобы оно явно преобразовывало значение выражения в тип string:

```
throw string( "help!" );
```

11.4.1. Спецификации исключений и указатели на функции

Спецификацию исключений можно задавать и при объявлении указателя на функцию. Например:

```
void (*pf)( int ) throw(string);
```

В этом объявлении говорится, что pf указывает на функцию, которая способна возбуждать только исключения типа string. Как и для объявлений функций, спецификации исключений в разных объявлениях одного и того же указателя не

```
extern void (*pf) ( int ) throw(string);

// ошибка: отсутствует спецификация исключения
```

суммируются, они должны быть одинаковыми:

```
void (*pf)( int );
```

При работе с указателем на функцию со спецификацией исключений есть ограничения на тип указателя, используемого в качестве инициализатора или стоящего в правой части присваивания. Спецификации исключений обоих указателей не обязаны быть идентичными. Однако на указатель-инициализатор она должна накладывать столь же или более строгие ограничения, чем на инициализируемый указатель (или тот, которому

```
void recoup( int, int ) throw(exceptionType);
void no_problem() throw();
void doit( int, int ) throw(string, exceptionType);

// правильно: ограничения, накладываемые на спецификации
// исключений recoup() и pf1, одинаковы
void (*pf1)( int, int ) throw(exceptionType) = &recoup;

// правильно: ограничения, накладываемые на спецификацию исключений
          no_problem(), более строгие,
// чем для pf2
void (*pf2)( ) throw(string) = &no_problem;

// ошибка: ограничения, накладываемые на спецификацию
// исключений doit(), менее строгие, чем для pf3
//
```

присваивается значение). Например:

```
void (*pf3)( int, int ) throw(string) = &doit;
```

Третья инициализация не имеет смысла. Объявление указателя гарантирует, что pf3 адресует функцию, которая может возбуждать только исключения типа string. Но doit() возбуждает также исключения типа exceptionType. Поскольку она не подходит под ограничения, накладываемые спецификацией исключений pf3, то не может служить корректным инициализатором для pf3, так что компилятор выдает ошибку.

Упражнение 11.9

В коде, разработанном для упражнения 11.8, измените объявление оператора operator[]() в классе IntArray, добавив спецификацию возбуждаемых им исключений. Модифицируйте программу так, чтобы operator[]() возбуждал исключение, не указанное в спецификации. Что при этом происходит?

Упражнение 11.10

Какие исключения может возбуждать функция, если ее спецификация исключений имеет вид throw()? А если у нее нет такой спецификации?

Упражнение 11.11

```
void example() throw(string);
(a) void (*pf1)() = example;
```

Какое из следующих присваиваний ошибочно? Почему?

```
(b) void (*pf2) throw() = example;
```

11.5. Исключения и вопросы проектирования

С обработкой исключений в программах C++ связано несколько вопросов. Хотя поддержка такой обработки встроена в язык, не стоит использовать ее везде. Обычно она применяется для обмена информацией об ошибках между независимо разработанными частями программы. Например, автор некоторой библиотеки может с помощью исключений сообщать пользователям об ошибках. Если библиотечная функция обнаруживает аномальную ситуацию, которую не способна обработать самостоятельно, она может возбудить исключение для уведомления вызывающей программы.

В нашем примере в библиотеке определен класс iStack и его функции-члены. Разумно предположить, что программист, кодировавший main(), где используется эта библиотека, не разрабатывал ее. Функции-члены класса iStack могут обнаружить, что операция pop() вызвана, когда стек пуст, или что операция push() вызвана, когда стек полон; однако разработчик библиотеки ничего не знал о программе, пользующейся его функциями, так что не мог разрешить проблему локально. Не сумев обработать ошибку внутри функций-членов, мы решили возбуждать исключения, чтобы известить вызывающую программу.

Хотя C++ поддерживает исключения, следует применять и другие методы обработки ошибок (например, возврат кода ошибки) – там, где это более уместно. Однозначного ответа на вопрос: “Когда ошибку следует трактовать как исключение?” не существует. Ответственность за решение о том, что считать исключительной ситуацией, возлагается на разработчика. Исключения – это часть интерфейса библиотеки, и решение о том, какие исключения она возбуждает, – важный аспект ее дизайна. Если библиотека предназначена для использования в программах, которые не должны аварийно

завершаться ни при каких обстоятельствах, то она обязана разбираться с аномалиями сама либо извещать о них вызывающую программу, передавая ей управление. Решение о том, какие ошибки следует обрабатывать как исключения, – трудная часть работы по проектированию библиотеки.

В нашем примере с классом `iStack` вопрос, должна ли функция `push()` возбуждать исключение, если стек полон, является спорным. Альтернативная и, по мнению многих, лучшая реализация `push()` – локальное решение проблемы: увеличение размера стека при его заполнении. В конце концов, единственное ограничение – это объем доступной программе памяти. Наше решение о возбуждении исключения при попытке поместить значение в полный стек, по-видимому, непродуманно. Можно переделать функцию-член

```
void iStack::push( int value )
{
   // если стек полон, увеличить размер вектора
   if ( full() )
      _stack.resize( 2 * _stack.size() );
   _stack[ _top++ ] = value;
}
```

`push()`, чтобы она в такой ситуации наращивала стек:

Аналогично следует ли функции `pop()` возбуждать исключение при попытке извлечь значение из пустого стека? Интересно отметить, что класс `stack` из стандартной библиотеки C++ (он рассматривался в главе 6) не возбуждает исключения в такой ситуации. Вместо этого постулируется, что поведение программы при попытке выполнения подобной операции не определено. Разрешить программе продолжать работу при обнаружении некорректного состояния признали возможным. Мы уже упоминали, что в разных библиотеках определены разные исключения. Не существует пригодного для всех случаев ответа на вопрос, что такое исключение.

Не все программы должны беспокоиться по поводу исключений, возбуждаемых библиотечными функциями. Хотя есть системы, для которых простой недопустим и которые, следовательно, должны обрабатывать все исключительные ситуации, не к каждой программе предъявляются такие требования. Обработка исключений предназначена в первую очередь для реализации отказоустойчивых систем. В этом случае решение о том, должна ли программа обрабатывать все исключения, возбуждаемые библиотеками, или может закончить выполнение аварийно, – это трудная часть процесса проектирования.

Еще один аспект проектирования программ заключается в том, что обработка исключений обычно структурирована. Как правило, программа строится из компонентов, и каждый компонент решает сам, какие исключения обрабатывать локально, а какие передавать на верхние уровни. Что мы понимаем под компонентом? Например, система анализа текстовых запросов, рассмотренная в главе 6, может быть разбита на три компонента, или слоя. Первый слой – это стандартная библиотека C++, которая обеспечивает базовые операции над строками, отображениями и т.д. Второй слой – это сама система анализа текстовых запросов, где определены такие функции, как `string_caps()` и `suffix_text()`, манипулирующие текстами и использующие стандартную библиотеку как основу. Третий слой – это программа, которая применяет нашу систему. Каждый компонент строится независимо и должен принимать решения о том, какие исключительные ситуации обрабатывать локально, а какие передавать на более высокий уровень.

Не все функции должны уметь обрабатывать исключения. Обычно try-блоки и ассоциированные с ними catch-обработчики применяются в функциях, являющихся точками входа в компонент. Catch-обработчики проектируются так, чтобы перехватывать те исключения, которые не должны попасть на верхние уровни программы. Для этого также используются спецификации исключений (см. раздел 11.4).

Мы расскажем о других аспектах проектирования программ, использующих исключения, в главе 19, после знакомства с классами и иерархиями классов.

12

12. Обобщенные алгоритмы

В нашу реализацию класса `Array` (см. главу 2) мы включили функции-члены для поддержки операций `min()`, `max()` и `sort()`. Однако в стандартном классе `vector` эти, на первый взгляд фундаментальные, операции отсутствуют. Для нахождения минимального или максимального значения элементов вектора следует вызвать один из *обобщенных алгоритмов*. Алгоритмами они называются потому, что реализуют такие распространенные операции, как `min()`, `max()`, `find()` и `sort()`, а обобщенными (generic) – потому, что применимы к различным контейнерным типам: векторам, спискам, массивам. Контейнер связывается с применяемым к нему обобщенным алгоритмом посредством пары итераторов (мы говорили о них в разделе 6.5), указывающих, какие элементы следует посетить при обходе контейнера. Специальные *объекты-функции* позволяют переопределить семантику операторов в обобщенных алгоритмах. Итак, в этой главе рассматриваются обобщенные алгоритмы, объекты-функции и итераторы.

12.1. Краткий обзор

Реализация обобщенного алгоритма не зависит от типа контейнера, поэтому одна основанная на шаблонах реализация может работать со всеми контейнерами, а равно и со встроенным типом массива. Рассмотрим алгоритм `find()`. Если коллекция не отсортирована, то, чтобы найти элемент, требуются лишь следующие общие шаги:

1. По очереди исследовать каждый элемент.
2. Если элемент равен искомому значению, то вернуть его позицию в коллекции.
3. В противном случае анализировать следующий элемент Повторять шаг 2, пока значение не будет найдено либо пока не будет просмотрена вся коллекция.
4. Если мы достигли конца коллекции и не нашли искомого, то вернуть некоторое значение, показывающее, что нужного элемента нет.

Алгоритм, как мы и утверждали, не зависит ни от типа контейнера, к которому применяется, ни от типа искомого значения, однако для его использования необходимы:

- способ обхода коллекции: переход к следующему элементу и распознавание того, что достигнут конец коллекции. При работе с встроенным типом массива мы решаем эту проблему, передавая два аргумента: указатель на первый элемент и число элементов, подлежащих обходу (в случае строк символов в стиле C передавать второй аргумент необязательно, так как конец строки обозначается двоичным нулем);
- умение сравнивать каждый элемент контейнера с искомым значением. Обычно это делается с помощью оператора равенства, ассоциированного со значениями типа, или путем передачи указателя на функцию, осуществляющую сравнение;
- некоторый обобщенный тип для представления позиции элемента внутри контейнера и специального признака на случай, если элемент не найден. Обычно мы возвращаем индекс элемента либо указатель на него. В ситуации, когда поиск неудачен, возвращается –1 вместо индекса или 0 вместо указателя.

Обобщенные алгоритмы решают первую проблему, обход контейнера, с помощью абстракции итератора – обобщенного указателя, поддерживающего оператор инкремента для доступа к следующему элементу, оператор разыменования для получения его значения и операторы равенства и неравенства для определения того, совпадают ли два итератора. Диапазон, к которому применяется алгоритм, помечается парой итераторов: `first` адресует первый элемент, а `last` – тот, который следует за последним. К самому элементу, адресованному итератором `last`, алгоритм не применяется; он служит *стражем*, прекращающим обход. Кроме того, `last` используется как возвращаемое значение с семантикой "отсутствует". Если же значение получено, то возвращается итератор, помечающий позицию найденного элемента.

Имеется по две версии каждого обобщенного алгоритма: в одной для сравнения применяется оператор равенства, а в другой – объект-функция или указатель на функцию, реализующую сравнение. (Объекты-функции рассматриваются в разделе 12.3.) Вот, например, реализация обобщенного алгоритма `find()`, в котором используется

```
template < class ForwardIterator, class Type >
ForwardIterator
find( ForwardIterator first, ForwardIterator last, Type value )
{
   for ( ; first != last; ++first )
            if ( value == *first )
          return first;
   return last;
}
```

оператор сравнения для типов хранимых в контейнере элементов:

`ForwardIterator` (однонаправленный итератор) – это один из пяти категорий итераторов, предопределенных в стандартной библиотеке. Он поддерживает чтение и запись адресуемого элемента. (Все пять категорий рассматриваются в разделе 12.4.)

Алгоритмы достигают независимости от типов за счет того, что никогда не обращаются к элементам контейнера непосредственно; доступ и обход элементов осуществляются только с помощью итераторов. Неизвестны ни фактический тип контейнера, ни даже то, является ли он контейнером или встроенным массивом. Для работы со встроенным типом массива обобщенному алгоритму можно передать не только обычные указатели, но и итераторы. Например, алгоритм `find()` для встроенного массива элементов типа `int` можно использовать так:

```
#include <algoritm>
#include <iostream>

int main()
{
   int search_value;
   int ia[ 6 ] = { 27, 210, 12, 47, 109, 83 };

   cout << "enter search value: ";
   cin >> search_value;

   int *presult = find( &ia[0], &ia[6], search_value );

   cout << "The value " << search_value
        << ( presult == &ia[6]
             ? " is not present" : " is present" )
   << endl;

}
```

Если возвращенный указатель равен адресу `&ia[6]` (который расположен за последним элементом массива), то поиск оказался безрезультатным, в противном случае значение найдено.

Вообще говоря, при передаче адресов элементов массива обобщенному алгоритму мы можем написать

```
int *presult = find( &ia[0], &ia[6], search_value );
```

или

```
int *presult = find( ia, ia+6, search_value );
```

Если бы мы хотели ограничиться лишь отрезком массива, то достаточно было бы модифицировать передаваемые алгоритму адреса. Так, при следующем обращении к `find()` просматриваются только второй и третий элементы (напомним, что элементы

```
// искать только среди элементов ia[1] и ia[2]
```

массива нумеруются с нуля):

```
int *presult = find( &ia[1], &ia[3], search_value );
```

А вот пример использования контейнера типа `vector` с алгоритмом `find()`:

```
#include <algorithm>
#include <vector>
#include <iostream>

int main()
{
   int search_value;
   int ia[ 6 ] = { 27, 210, 12, 47, 109, 83 };
   vector<int> vec( ia, ia+6 );

   cout << "enter search value: ";
   cin >> search_value;

   vector<int>::iterator presult;
   presult = find( vec.begin(), vec.end(), search_value );

   cout << "The value " << search_value
        << ( presult == vec.end()
            ? " is not present" : " is present" )
        << endl;
}
```

```
#include <algorithm>
#include <list>
#include <iostream>
int main()
{
   int search_value;
   int ia[ 6 ] = { 27, 210, 12, 47, 109, 83 };
   list<int> ilist( ia, ia+6 );

   cout << "enter search value: ";
   cin >> search_value;

   list<int>::iterator presult;
   presult = find( ilist.begin(), ilist.end(), search_value );

   cout << "The value " << search_value
        << ( presult == ilist.end()
            ? " is not present" : " is present" )
   << endl;
}
```

`find()` можно применить и к списку:

(В следующем разделе мы обсудим построение программы, в которой используются различные обобщенные алгоритмы, а затем рассмотрим объекты-функции. В разделе 12.4 мы подробнее расскажем об итераторах. Развернутое введение в обобщенные алгоритмы – предмет раздела 12.5, а их детальное обсуждение и иллюстрация применения вынесено в Приложение. В конце главы речь пойдет о случаях, когда применение обобщенных алгоритмов неуместно.)

Упражнение 12.1

Обобщенные алгоритмы критикуют за то, что при всей элегантности дизайна проверка корректности возлагается на программиста. Например, если передан неверный итератор или пара итераторов, помечающая неверный диапазон, то поведение программы не

определено. Вы согласны с такой критикой? Следует ли оставить применение обобщенных алгоритмов только наиболее квалифицированным специалистам? Может быть, нужно запретить использование потенциально опасных конструкций, таких, как обобщенные алгоритмы, указатели и явные приведения типов?

12.2. Использование обобщенных алгоритмов

Допустим, мы задумали написать книжку для детей и хотим понять, какой словарный состав наиболее подходит для такой цели. Чтобы ответить на этот вопрос, нужно прочитать несколько детских книг, сохранить текст в отдельных векторах строк (см. раздел 6.7) и подвергнуть его следующей обработке:

1. Создать копию каждого вектора.
2. Слить все векторы в один.
3. Отсортировать его в алфавитном порядке.
4. Удалить все дубликаты.
5. Снова отсортировать, но уже по длине слов.
6. Подсчитать число слов, длина которых больше шести знаков (предполагается, что длина – это некоторая мера сложности, по крайней мере, в терминах словаря).
7. Удалить семантически нейтральные слова (например, союзы and (и), if (если), or (или), but (но) и т.д.).
8. Напечатать получившийся вектор.

На первый взгляд, задача на целую главу. Но с помощью обобщенных алгоритмов мы решим ее в рамках одного подраздела.

Аргументом нашей функции является вектор из векторов строк. Мы принимаем указатель

```
#include <vector>
#include <string>

typedef vector<string, allocator> textwords;
void process_vocab( vector<textwords, allocator> *pvec )
{
   if ( ! pvec ) {
      // выдать предупредительное сообщение
      return;
   }

   // ...
```

на него, проверяя, не является ли он нулевым:

```
}
```

Нужно создать один вектор, включающий все элементы исходных векторов. Это делается с помощью обобщенного алгоритма `copy()` (для его использования необходимо включить заголовочные файлы `algorithm` и `iterator`):

```
#include <algorithm>
#include <iterator>

void process_vocab( vector<textwords, allocator> *pvec )
{
   // ...
   vector< string > texts;

   vector<textwords, allocator>::iterator iter = pvec->begin();
   for ( ; iter != pvec->end(); ++iter )
      copy( (*iter).begin(), (*iter).end(), back_inserter( texts ));

   // ...

}
```

Первыми двумя аргументами алгоритма `copy()` являются итераторы, ограничивающие диапазон подлежащих копированию элементов. Третий аргумент – это итератор, указывающий на место, куда надо копировать элементы. `back_inserter` называется *адаптером итератора*; он позволяет вставлять элементы в конец вектора, переданного ему в качестве аргумента. (Подробнее мы рассмотрим адаптеры итераторов в разделе 12.4.).

Алгоритм `unique()` удаляет из контейнера дубликаты, расположенные рядом. Если дана последовательность `01123211`, то результатом будет `012321`, а не `0123`. Чтобы получить вторую последовательность, необходимо сначала отсортировать вектор с помощью алгоритма `sort()`; тогда из последовательности `01111223` получится `0123`. (Хотя на самом деле получится `01231223`.)

`unique()` не изменяет размер контейнера. Вместо этого каждый уникальный элемент помещается в очередную свободную позицию, начиная с первой. В нашем примере физический результат – это последовательность `01231223`; остаток `1223` – это, так сказать, “отходы” алгоритма. `unique()` возвращает итератор, указывающий на начало этого остатка. Как правило, этот итератор затем передается алгоритму `erase()` для удаления ненужных элементов. (Поскольку встроенный массив не поддерживает операции `erase()`, то семейство алгоритмов `unique()` в меньшей степени подходит для работы с ним.) Вот соответствующий фрагмент функции:

```
void process_vocab( vector<textwords, allocator> *pvec )
{
   // ...
   // отсортировать вектор texts
   sort( texts.begin(), texts.end() );

   // удалить дубликаты
   vector<string, allocator>::iterator it;
   it = unique( texts.begin(), texts.end() );
   texts.erase( it, texts.end() );

   // ...

}
```

Ниже приведен результат печати вектора `texts`, объединяющего два небольших текстовых файла, после применения `sort()`, но до применения `unique()`:

```
a a a a alice alive almost
alternately ancient and and and and and and
and as asks at at beautiful becomes bird
bird blows blue bounded but by calling coat
daddy daddy daddy dark darkened darkening distant each
either emma eternity falls fear fiery fiery flight
flowing for grow hair hair has he heaven,
held her her her her him him home
houses i immeasurable immensity in in in in
inexpressibly is is is it it it its
journeying lands leave leave life like long looks
magical mean more night, no not not not
now now of of on one one one
passion puts quite red rises row same says
she she shush shyly sight sky so so
star star still stone such tell tells tells
that that the the the the the the
the there there thing through time to to
to to trees unravel untamed wanting watch what
when wind with with you you you you
your your
```

После применения `unique()` и последующего вызова `erase()` вектор `texts` выглядит следующим образом:

```
a alice alive almost alternately ancient
and as asks at beautiful becomes bird blows
blue bounded but by calling coat daddy dark
darkened darkening distant each either emma eternity falls
fear fiery flight flowing for grow hair has
he heaven, held her him home houses i
immeasurable immensity in inexpressibly is it its journeying
lands leave life like long looks magical mean
more night, no not now of on one
passion puts quite red rises row same says
she shush shyly sight sky so star still
stone such tell tells that the there thing
through time to trees unravel untamed wanting watch
what when wind with you your
```

Следующая наша задача – отсортировать строки по длине. Для этого мы воспользуемся не алгоритмом `sort()`, а алгоритмом `stable_sort()`, который сохраняет относительные положения равных элементов. В результате для элементов равной длины сохраняется алфавитный порядок. Для сортировки по длине мы применим собственную операцию

```
bool less_than( const string & s1, const string & s2 )
{
   return s1.size() < s1.size();
}

void process_vocab( vector<textwords, allocator> *pvec )
{
   // ...
   // отсортировать элементы вектора texts по длине,
   // сохранив также прежний порядок
   stable_sort( texts.begin(), texts.end(), less_than );

   // ...
```

сравнения “меньше”. Один из возможных способов таков:

```
}
```

Нужный результат при этом достигается, но эффективность существенно ниже, чем хотелось бы. less_than() реализована в виде одной инструкции. Обычно она вызывается как встроенная (inline) функция. Но, передавая указатель на нее, мы не даем компилятору сделать ее встроенной. Способ, позволяющий добиться этого, –применение

```
// объект-функция - операция реализована с помощью перегрузки
// оператора operator()
class LessThan {
public:
   bool operator()( const string & s1, const string & s2 )
                  { return s1.size() < s2.size(); }
```

объекта-функции:

```
};
```

Объект-функция – это класс, в котором перегружен оператор вызова operator(). В теле этого оператора и реализуется логика функции, в данном случае сравнение “меньше”. Определение оператора вызова выглядит странно из-за двух пар скобок. Запись

```
operator()
```

говорит компилятору, что мы перегружаем оператор вызова. Вторая пара скобок

```
( const string & s1, const string & s2 )
```

задает передаваемые ему формальные параметры. Если сравнить это определение с предыдущим определением функции less_than(), мы увидим, что, за исключением замены less_than на operator(), они совпадают.

Объект-функция определяется так же, как обычный объект класса (правда, в данном случае нам не понадобился конструктор: нет членов, подлежащих инициализации):

```
LessThan lt;
```

Для вызова экземпляра перегруженного оператора мы применяем оператор вызова к

```
string st1( "shakespeare" );
string st2( "marlowe" );

// вызывается lt.operator()( st1, st2 );
```

нашему объекту класса, передавая необходимые аргументы. Например:

```
bool is_shakespeare_less = lt( st1, st2 );
```

Ниже показана исправленная функция process_vocab(), в которой алгоритму stable_sort() передается безымянный объект-функция LessThan():

```
void process_vocab( vector<textwords, allocator> *pvec )
{
   // ...
   stable_sort( texts.begin(), texts.end(), LessThan() );

   // ...
}
```

Внутри `stable_sort()` перегруженный оператор вызова подставляется в текст программы как встроенная функция. (В качестве третьего аргумента `stable_sort()` может принимать как указатель на функцию `less_than()`, так и объект класса `LessThan`, поскольку аргументом является параметр-тип шаблона. Подробнее об объектах-функциях мы расскажем в разделе 12.3.)

Вот результат применения `stable_sort()` к вектору `texts`:

```
a i
as at by he in is it no
of on so to and but for has
her him its not now one red row
she sky the you asks bird blue coat
dark each emma fear grow hair held home
life like long mean more puts same says
star such tell that time what when wind
with your alice alive blows daddy falls fiery
lands leave looks quite rises shush shyly sight
still stone tells there thing trees watch almost
either flight houses night, ancient becomes bounded calling
distant flowing heaven, magical passion through unravel untamed
wanting darkened eternity beautiful darkening immensity journeying alternately
immeasurable inexpressibly
```

Подсчитать число слов, длина которых больше шести символов, можно с помощью обобщенного алгоритма `count_if()` и еще одного объекта-функции – `GreaterThan`. Этот объект чуть сложнее, так как позволяет пользователю задать размер, с которым производится сравнение. Мы сохраняем размер в члене класса и инициализируем его с

```
#include <iostream>

class GreaterThan {
public:
   GreaterThan( int size = 6 ) : _size( size ){}
   int size() { return _size; }

   bool operator()( const string & s1 )
      { return s1.size() > 6; }

private:
   int _size;
```

помощью конструктора (по умолчанию – значением 6):

```
};
```

Использовать его можно так:

```
void process_vocab( vector<textwords, allocator> *pvec )
{
   // ...
   // подсчитать число строк, длина которых больше 6

   int cnt = count_if( texts.begin(), texts.end(),
                       GreaterThan() );

   cout << "Number of words greater than length six are "
        << cnt << endl;

   // ...

}
```

Этот фрагмент программы выводит такую строку:

```
Number of words greater than length six are 22
```

Алгоритм `remove()` ведет себя аналогично `unique()`: он тоже не изменяет размер контейнера, а просто разделяет элементы на те, что следует оставить (копируя их по очереди в начало контейнера), и те, что следует удалить (перемещая их в конец контейнера). Вот как можно воспользоваться им для исключения из коллекции слов,

```
void process_vocab( vector<textwords, allocator> *pvec )
{
   // ...
   static string rw[] = { "and", "if", "or", "but", "the" };
   vector< string > remove_words( rw, rw+5 );

   vector< string >::iterator it2 = remove_words.begin();
   for ( ; it2 != remove_words.end(); ++it2 ) {
      // просто для демонстрации другой формы count()
           int cnt = count( texts.begin(), texts.end(), *it2 );
      cout << cnt << " instances removed:  "
           << (*it2) << endl;

      texts.erase(
           remove(texts.begin(),texts.end(),*it2 ),
           texts.end()
           );
   }

   // ...
```

которые мы не хотим сохранять:

```
}
```

Результат применения `remove()`:

```
1 instances removed:  and
0 instances removed:  if
0 instances removed:  or
1 instances removed:  but
1 instances removed:  the
```

Теперь нам нужно распечатать содержимое вектора. Можно обойти все элементы и вывести каждый по очереди, но, поскольку при этом обобщенные алгоритмы не используются, мы считаем такое решение неподходящим. Вместо этого проиллюстрируем работу алгоритма `for_each()` для вывода всех элементов вектора. `for_each()` применяет указатель на функцию или объект-функцию к каждому элементу контейнера из диапазона, ограниченного парой итераторов. В нашем случае объект-функция

```
class PrintElem {
public:
   PrintElem( int lineLen = 8 )
      : _line_length( lineLen ), _cnt( 0 )
   {}

   void operator()( const string &elem )
   {
      ++_cnt;
      if ( _cnt % _line_length == 0 )
         { cout << '\n'; }

      cout << elem << " ";
   }

private:
   int _line_length;
   int _cnt;
```

`PrintElem` копирует один элемент в стандартный вывод:

```
void process_vocab( vector<textwords, allocator> *pvec )
{
   // ...

   for_each( texts.begin(), texts.end(), PrintElem() );

};
}
```

Вот и все. Мы получили законченную программу, для чего пришлось лишь последовательно записать обращения к нескольким обобщенным алгоритмам. Для удобства мы приводим ниже полный листинг вместе с функцией `main()` для ее тестирования (здесь используются специальные типы итераторов, которые будут обсуждаться только в разделе 12.4). Мы привели текст реально исполнявшегося кода, который не полностью удовлетворяет стандарту C++. В частности, в нашем распоряжении были лишь устаревшие реализации алгоритмов `count()` и `count_if()`, которые не возвращают результат, а требуют передачи дополнительного аргумента для вычисленного значения. Кроме того, библиотека `iostream` отражает предшествующую принятию стандарта реализацию, в которой требуется заголовочный файл `iostream.h`.

```
#include <vector>
#include <string>
#include <algorithm>
#include <iterator>

// предшествующий принятию стандарта синтаксис <iostream>
#include <iostream.h>

class GreaterThan {
public:
   GreaterThan( int size = 6 ) : _size( sz ){}
   int size() { return _size; }

   bool operator()(const string &s1)
                  { return s1.size() > _size; }
private:
   int _size;
};

class PrintElem {
public:

   PrintElem( int lineLen = 8 )
      : _line_length( lineLen ), _cnt( 0 )
   {}

   void operator()( const string &elem )
   {
      ++_cnt;
      if ( _cnt % _line_length == 0 )
         { cout << '\n'; }

      cout << elem << " ";
   }

private:
   int _line_length;
   int _cnt;

};
```

```
public:
   bool operator()( const string & s1,
                    const string & s2 )
   { return s1.size() < s2.size();   }
};

typedef vector<string, allocator> textwords;
void process_vocab( vector<textwords, allocator> *pvec )
{
   if ( ! pvec ) {
           // вывести предупредительное сообщение
     return;
   }

   vector< string, allocator > texts;

   vector<textwords, allocator>::iterator iter;
   for ( iter = pvec->begin() ; iter != pvec->end(); ++iter )
      copy( (*iter).begin(), (*iter).end(),
              back_inserter( texts ));

   // отсортировать вектор texts
   sort( texts.begin(), texts.end() );

   // теперь посмотрим, что получилось
   for_each( texts.begin(), texts.end(), PrintElem() );

   cout << "\n\n";   // разделить части выведенного текста

   // удалить дубликаты
   vector<string, allocator>::iterator it;
   it = unique( texts.begin(), texts.end() );
   texts.erase( it, texts.end() );

   // посмотрим, что осталось
   for_each( texts.begin(), texts.end(), PrintElem() );
   cout << "\n\n";

   // отсортировать элементы
   // stable_sort сохраняет относительный порядок равных элементов
   stable_sort( texts.begin(), texts.end(), LessThan() );
   for_each( texts.begin(), texts.end(), PrintElem() );

   cout << "\n\n";

   // подсчитать число строк, длина которых больше 6
   int cnt = 0;

   // устаревшая форма count - в стандарте используется другая
   count_if( texts.begin(), texts.end(), GreaterThan(), cnt );

   cout << "Number of words greater than length six are "
        << cnt << endl;

   static string rw[] = { "and", "if", "or", "but", "the" };
   vector<string,allocator> remove_words( rw, rw+5 );

   vector<string, allocator>::iterator it2 = remove_words.begin();

   for ( ; it2 != remove_words.end(); ++it2 )
   {
      int cnt = 0;

      // устаревшая форма count - в стандарте используется другая
      count( texts.begin(), texts.end(), *it2, cnt );

      cout << cnt    << " instances removed:  "
           << (*it2) << endl;

      texts.erase(
         remove(texts.begin(),texts.end(),*it2),
                texts.end()
      );
   }
   cout << "\n\n";
   for_each( texts.begin(), texts.end(), PrintElem() );
}
```

```
}
```

Упражнение 12.2

Длина слова – не единственная и, вероятно, не лучшая мера трудности текста. Другой возможный критерий – это длина предложения. Напишите программу, которая читает текст из файла либо со стандартного ввода, строит вектор строк для каждого предложения и передает его алгоритму `count()`. Выведите предложения в порядке сложности. Любопытный способ сделать это – сохранить каждое предложение как одну большую строку во втором векторе строк, а затем передать этот вектор алгоритму `sort()` вместе с объектом-функцией, который считает, что чем строка короче, тем она меньше. (Более подробно с описанием конкретного обобщенного алгоритма, а также с иллюстрацией его применения вы может ознакомиться в Приложении, где все алгоритмы перечислены в алфавитном порядке.)

Упражнение 12.3

Более надежную оценку уровня трудности текста дает анализ структурной сложности предложений. Пусть каждой запятой присваивается 1 балл, каждому двоеточию или точке с запятой – 2 балла, а каждому тире – 3 балла. Модифицируйте программу из упражнения 12.2 так, чтобы она подсчитывала сложность каждого предложения. Воспользуйтесь алгоритмом `count_if()` для нахождения каждого из знаков препинания в векторе предложений. Выведите предложения в порядке сложности.

12.3. Объекты-функции

Наша функция `min()` дает хороший пример как возможностей, так и ограничений

```
template <typename Type>
const Type&
min( const Type *p, int size )
{
   Type minval = p[ 0 ];
   for ( int ix = 1; ix < size; ++ix )
      if ( p[ ix ] < minval )
         minval = p[ ix ];
      return minval;
```

механизма шаблонов:

```
}
```

Достоинство этого механизма – возможность определить единственный шаблон `min()`, который конкретизируется для бесконечного множества типов. Ограничение же заключается в том, что даже при такой конкретизации `min()` будет работать не со всеми.

Это ограничение вызвано использованием оператора “меньше”: в некоторых случаях базовый тип его не поддерживает. Так, класс изображения `Image` может и не предоставлять реализации такого оператора, но мы об этом не знаем и пытаемся найти минимальный кадр анимации в данном массиве изображений. Однако попытка конкретизировать `min()` для такого массива приведет к ошибке компиляции:

```
error: invalid types applied to the < operator: Image < Image
(ошибка: оператор < применен к некорректным типам: Image < Image)
```

Возможна и другая ситуация: оператор “меньше” существует, но имеет неподходящую семантику. Например, если мы хотим найти наименьшую строку, но при этом принимать во внимание только буквы, не учитывая регистр, то такой реализованный в классе оператор не даст нужного результата.

Традиционное решение состоит в том, чтобы параметризовать оператор сравнения. В данном случае это можно сделать, объявив указатель на функцию, принимающую два

```
template < typename Type,
           bool (*Comp)(const Type&, const Type&)>
const Type&
min( const Type *p, int size, Comp comp )
{
   Type minval = p[ 0 ];
   for ( int ix = 1; ix < size; ++ix )
      if ( Comp( p[ ix ] < minval ))
         minval = p[ ix ];
      return minval;
}
```

аргумента и возвращающую значение типа bool:

Такое решение вместе с нашей первой реализацией на основе встроенного оператора “меньше” обеспечивает универсальную поддержку для любого типа, включая и класс Image, если только мы придумаем подходящую семантику для сравнения двух изображений. Основной недостаток указателя на функцию связан с низкой эффективностью, так как косвенный вызов не дает воспользоваться преимуществами встроенных функций.

Альтернативная стратегия параметризации заключается в применении объекта-функции вместо указателя (примеры мы видели в предыдущем разделе). Объект-функция – это класс, перегружающий оператор вызова (operator()). Такой оператор инкапсулирует семантику обычного вызова функции. Объект-функция, как правило, передается обобщенному алгоритму в качестве аргумента, хотя можно определять и независимые объекты-функции. Например, если бы был определен объект-функция AddImages, который принимает два изображения, объединяет их некоторым образом и возвращает новое изображение, то мы могли бы объявить его следующим образом:

```
AddImages AI;
```

Чтобы объект-функция удовлетворял нашим требованиям, мы применяем оператор

```
Image im1("foreground.tiff"), im2("background.tiff");
// ...

// вызывает Image AddImages::operator()(const Image1&, const Image2&);
```

вызова, предоставляя необходимые операнды в виде объектов класса Image:

```
Image new_image = AI (im1, im2 );
```

У объекта-функции есть два преимущества по сравнению с указателем на функцию. Во-первых, если перегруженный оператор вызова – это встроенная функция, то компилятор может выполнить ее подстановку, обеспечивая значительный выигрыш в производительности. Во-вторых, объект-функция способен содержать произвольное

количество дополнительных данных, например кэш или информацию, полезную для выполнения текущей операции.

Ниже приведена измененная реализация шаблона `min()` (отметим, что это объявление

```
template < typename Type,
           typename Comp >
const Type&
min( const Type *p, int size, Comp comp )
{
   Type minval = p[ 0 ];
   for ( int ix = 1; ix < size; ++ix )
      if ( Comp( p[ ix ] < minval ))
         minval = p[ ix ];
      return minval;
```

допускает также и передачу указателя на функцию, но без проверки прототипа):

```
}
```

Как правило, обобщенные алгоритмы поддерживают обе формы применения операции: как использование встроенного (или перегруженного) оператора, так и применение указателя на функцию либо объекта-функции.

Есть три источника появления объектов-функций:

1. из набора предопределенных арифметических, сравнительных и логических объектов-функций стандартной библиотеки;
2. из набора предопределенных адаптеров функций, позволяющих специализировать или расширять предопределенные (или любые другие) объекты-функции;
3. определенные нами собственные объекты-функции для передачи обобщенным алгоритмам. К ним можно применять и адаптеры функций.

В этом разделе мы рассмотрим все три источника объектов-функций.

12.3.1. Предопределенные объекты-функции

Предопределенные объекты-функции подразделяются на арифметические, логические и сравнительные. Каждый объект – это шаблон класса, параметризованный типами операндов. Для использования любого из них необходимо включить заголовочный файл:

```
#include <functional>
```

Например, объект-функция, поддерживающий сложение, – это шаблон класса с именем `plus`. Для определения экземпляра, способного складывать два целых числа, нужно

```
#include <functional>
```

написать:

```
plus< int > intAdd;
```

Для выполнения операции сложения мы применяем перегруженный оператор вызова к `intAdd` точно так же, как и к классу `AddImage` в предыдущем разделе:

```
int ival1 = 10, ival2 = 20;

// эквивалентно int sum = ival1 + ival2;
int sum = intAdd( ival1, ival2 );
```

Реализация шаблона класса `plus` вызывает оператор сложения, ассоциированный с типом своего параметра – `int`. Этот и другие предопределенные объекты-функции применяются прежде всего в качестве аргументов обобщенных алгоритмов и обычно замещают подразумеваемую по умолчанию операцию. Например, по умолчанию алгоритм `sort()` располагает элементы контейнера в порядке возрастания с помощью оператора “меньше” базового типа. Для сортировки по убыванию мы передаем

```
vector< string > svec;
// ...
```

предопределенный шаблон класса `greater`, который вызывает оператор “больше”:

```
sort( svec.begin(), svec.end(), greater<string>() );
```

Предопределенные объекты-функции перечислены в следующих разделах и разбиты на категории: арифметические, логические и сравнительные. Применение каждого из них иллюстрируется как в качестве именованного, так и в качестве безымянного объекта, передаваемого функции. Мы пользуемся следующими определениями объектов, включая и определение простого класса (перегрузка операторов подробно рассматривается в главе

```
class Int {
public:
   Int( int ival = 0 ) : _val( ival ) {}

   int operator-()          { return -_val;         }
   int operator%(int ival)  { return -_val % ival;  }

   bool operator<(int ival) { return -_val < ival;  }
   bool operator!()         { return -_val == 0;    }
private:
   int _val;
};

vector< string > svec;
string  sval1, sval2, sres;
complex cval1, cval2, cres;
int     ival1, ival2, ires;
Int     Ival1, Ival2, Ires;
```

15):

```
double  dval1, dval2, dres;
```

Кроме того, мы определяем два шаблона функций, которым передаем различные безымянные объекты-функции:

```
template <class FuncObject, class Type>
   Type UnaryFunc( FuncObject fob, const Type &val )
      { return fob( val ); }

template <class FuncObject, class Type>
   Type BinaryFunc( FuncObject fob,
                    const Type &val1, const Type &val2 )
      { return fob( val1, val2 ); }
```

12.3.2. Арифметические объекты-функции

Предопределенные арифметические объекты-функции поддерживают операции сложения, вычитания, умножения, деления, взятия остатка и вычисления противоположного по знаку значения. Вызываемый оператор – это экземпляр, ассоциированный с типом Type. Если тип является классом, предоставляющим перегруженную реализацию оператора, то именно эта реализация и вызывается.

```
plus<string> stringAdd;

// вызывается string::operator+()
sres = stringAdd( sval1, sval2 );
```

- Сложение: plus<Type>

```
dres = BinaryFunc( plus<double>(), dval1, dval2 );
```

```
minus<int> intSub;
ires = intSub( ival1, ival2 );
```

- Вычитание: minus<Type>

```
dres = BinaryFunc( minus<double>(), dval1, dval2 );
```

```
multiplies<complex> complexMultiplies;
cres = complexMultiplies( cval1, cval2 );
```

- Умножение: multiplies<Type>

```
dres = BinaryFunc( multiplies<double>(), dval1, dval2 );
```

```
divides<int> intDivides;
ires = intDivides( ival1, ival2 );
```

- Деление: divides<Type>

```
dres = BinaryFunc( divides<double>(), dval1, dval2 );
```

- Взятие остатка: modulus<Type>

```
modulus<Int> IntModulus;
Ires = IntModulus( Ival1, Ival2 );
```

```
ires = BinaryFunc( modulus<int>(), ival1, ival2 );
```

```
negate<int> intNegate;
ires = intNegate( ires );
```

- Вычисление противоположного значения: negate<Type>

```
Ires = UnaryFunc( negate<Int>(), Ival1 );
```

12.3.3. Сравнительные объекты-функции

Сравнительные объекты-функции поддерживают операции равенства, неравенства, больше, больше или равно, меньше, меньше или равно.

```
equal_to<string> stringEqual;
sres = stringEqual( sval1, sval2 );
ires = count_if( svec.begin(), svec.end(),
```

- Равенство: equal_to<Type>

```
                equal_to<string>(), sval1 );
```

```
not_equal_to<complex> complexNotEqual;
cres = complexNotEqual( cval1, cval2 );
ires = count_if( svec.begin(), svec.end(),
```

- Неравенство: not_equal_to<Type>

```
                not_equal_to<string>(), sval1 );
```

```
greater<int> intGreater;
ires = intGreater( ival1, ival2 );
ires = count_if( svec.begin(), svec.end(),
```

- Больше: greater<Type>

```
                greater<string>(), sval1 );
```

```
greater_equal<double> doubleGreaterEqual;
dres = doubleGreaterEqual( dval1, dval2 );
ires = count_if( svec.begin(), svec.end(),
```

- Больше или равно: greater_equal<Type>

```
                greater_equal <string>(), sval1 );
```

```
less<Int> IntLess;
Ires = IntLess( Ival1, Ival2 );
ires = count_if( svec.begin(), svec.end(),
```

- Меньше: less<Type>

```
                less<string>(), sval1 );
```

```
less_equal<int> intLessEqual;
ires = intLessEqual( ival1, ival2 );
ires = count_if( svec.begin(), svec.end(),
```

- Меньше или равно: less_equal<Type>

```
                less_equal<string>(), sval1 );
```

12.3.4. Логические объекты-функции

Логические объекты-функции поддерживают операции “логическое И” (возвращает true, если оба операнда равны true, – применяет оператор &&, ассоциированный с типом Type), “логическое ИЛИ” (возвращает true, если хотя бы один из операндов равен true, – применяет оператор ||, ассоциированный с типом Type) и “логическое НЕ” (возвращает true, если операнд равен false, – применяет оператор !, ассоциированный с типом Type)

```
logical_and<int> intAnd;
ires = intLess( ival1, ival2 );
```

- Логическое И: logical_and<Type>

```
dres = BinaryFunc( logical_and<double>(), dval1, dval2 );
```

```
logical_or<int> intSub;
ires = intSub( ival1, ival2 );
```

- Логическое ИЛИ: logical_or<Type>

```
dres = BinaryFunc( logical_or<double>(), dval1, dval2 );
```

```
logical_not<Int> IntNot;
ires = IntNot( Ival1, Ival2 );
```

- Логическое НЕ: logical_not<Type>

```
dres = UnaryFunc( logical_or<double>(), dval1 );
```

12.3.5. Адаптеры функций для объектов-функций

В стандартной библиотеке имеется также ряд адаптеров функций, предназначенных для специализации и расширения как унарных, так и бинарных объектов-функций. Адаптеры – это специальные классы, разбитые на следующие две категории:

- связыватели (binders). Это адаптеры, преобразующие бинарный объект-функцию в унарный объект, связывая один из аргументов с конкретным значением. Например, для подсчета в контейнере всех элементов, которые меньше или равны 10, следует передать алгоритму `count_if()` объект-функцию `less_equal`, один из аргументов которого равен `10`. В следующем разделе мы покажем, как это сделать;
- отрицатели (negators). Это адаптеры, изменяющие значение истинности объекта-функции на противоположное. Например, для подсчета всех элементов внутри контейнера, которые больше 10, мы могли бы передать алгоритму `count_if()` отрицатель объекта-функции `less_equal`, один из аргументов которого равен `10`. Конечно, в данном случае проще передать связыватель объекта-функции `greater`, ограничив один из аргументов со значением `10`.

В стандартную библиотеку входит два предопределенных адаптера-связывателя: `bind1st` и `bind2nd`, причем `bind1st` связывает некоторое значение с первым аргументом бинарного объекта-функции, а `bind2nd` – со вторым. Например, для подсчета внутри контейнера всех элементов, которые меньше или равны `10`, мы могли бы передать

```
count_if( vec.begin(), vec.end(),
```

алгоритму `count_if()` следующее:

```
          bind2nd( less_equal<int>(), 10 ));
```

В стандартной библиотеке также есть два предопределенных адаптера-отрицателя: `not1` и `not2`. `not1` инвертирует значение истинности унарного предиката, являющегося объектом-функцией, а `not2` – значение бинарного предиката. Для отрицания рассмотренного выше связывателя объекта-функции `less_equal` можно написать

```
count_if( vec.begin(), vec.end(),
```

следующее:

```
          not1( bind2nd( less_equal<int>(), 10 )));
```

Другие примеры использования связывателей и отрицателей приведены в Приложении, вместе с примерами использования каждого алгоритма.

12.3.6. Реализация объекта-функции

При реализации программы в разделе 12.2 нам уже приходилось определять ряд объектов-функций. В этом разделе мы изучим необходимые шаги и возможные вариации при определении класса объекта-функции. (В главе 13 определение класса рассматривается детально; в главе 15 обсуждается перегрузка операторов.)

В самой простой форме определение класса объекта-функции сводится к перегрузке оператора вызова. Вот, например, унарный объект-функция, определяющий, что

```
// простейшая форма класса объекта-функции
class less_equal_ten {
public:
   bool operator() ( int val )
        { return val <= 10; }
};
```

некоторое значение меньше или равно 10:

Теперь такой объект-функцию можно использовать точно так же, как предопределенный. Вызов алгоритма `count_if()` с помощью нашего объекта-функции выглядит следующим образом:

```
count_if( vec.begin(), vec.end(), less_equal_ten() );
```

Разумеется, возможности этого класса весьма ограничены. Попробуем применить

```
count_if( vec.begin(), vec.end(),
          not1(less_equal_then ()));
```

отрицатель, чтобы подсчитать, сколько в контейнере элементов, больших 10:

или обобщить реализацию, разрешив пользователю задавать значение, с которым надо сравнивать каждый элемент контейнера. Для этого достаточно ввести в класс член для хранения такого значения и реализовать конструктор, инициализирующий данный член

```
class less_equal_value {
public:
   less_equal_value( int val ) : _val( val ) {}
   bool operator() ( int val ) { return val <= _val; }

private:
   int _val;
};
```

указанной пользователем величиной:

Новый объект-функция применяется для задания произвольного целого значения. Например, при следующем вызове подсчитывается число элементов, меньших или равных 25:

```
count_if( vec.begin(), vec.end(), less_equal_value( 25 ));
```

Разрешается реализовать класс и без конструктора, если параметризовать его значением, с которым производится сравнение:

```
template < int _val >
class less_equal_value {
public:
   bool operator() ( int val ) { return val <= _val; }
};
```

Вот как надо было бы вызвать такой класс для подсчета числа элементов, меньших или равных 25:

```
count_if( vec.begin(), vec.end(), less_equal_value<25>());
```

(Другие примеры определения собственных объектов-функций можно найти в Приложении.)

Упражнение 12.4

Используя предопределенные объекты-функции и адаптеры, создайте объекты-функции для решения следующих задач:

(a) Найти все значения, большие или равные 1024.

(b) Найти все строки, не равные "pooh".

(c) Умножить все значения на 2.

Упражнение 12.5

Определите объект-функцию для возврата среднего из трех объектов. Определите функцию для выполнения той же операции. Приведите примеры использования каждого объекта непосредственно и путем передачи его функции. Покажите, в чем сходство и различие этих решений.

12.4. Еще раз об итераторах

Следующая реализация шаблона функции не компилируется. Можете ли вы сказать,

```
// в таком виде это не компилируется
template < typename type >
int
count( const vector< type > &vec, type value )
{
   int count = 0;

   vector< type >::iterator iter = vec.begin();
   while ( iter != vec.end() )
      if ( *iter == value )
         ++count;

   return count;
}
```

почему?

Проблема в том, что у ссылки vec есть спецификатор const, а мы пытаемся связать с ней итератор без такого спецификатора. Если бы это было разрешено, то ничто не помешало бы нам модифицировать с помощью этого итератора элементы вектора. Для

предотвращения подобной ситуации язык требует, чтобы итератор, связанный с const-

```
// правильно: это компилируется без ошибок
```

вектором, был константным. Мы можем сделать это следующим образом:

```
vector< type>::const_iterator iter = vec.begin();
```

Требование, чтобы с const-контейнером был связан только константный итератор, аналогично требованию о том, чтобы const-массив адресовался только константным указателем. В обоих случаях это вызвано необходимостью гарантировать, что содержимое const-контейнера не будет изменено.

Операции begin() и end() перегружены и возвращают константный или неконстантный итератор в зависимости от наличия спецификатора const в объявлении контейнера. Если

```
vector< int > vec0;
```

дана такая пара объявлений:

```
const vector< int > vec1;
```

то при обращениях к begin() и end() для vec0 будет возвращен неконстантный, а для

```
vector< int >::iterator iter0 = vec0.begin();
```

vec1 – константный итератор:

```
vector< int >::const_iterator iter1 = vec1.begin();
```

Разумеется, присваивание константному итератору неконстантного разрешено всегда.

```
// правильно: инициализация константного итератора неконстантным
```

Например:

```
vector< int >::const_iterator iter2 = vec0.begin();
```

12.4.1. Итераторы вставки

Вот еще один фрагмент программы, в котором есть тонкая, но серьезная ошибка. Видите

```
int ia[] = { 0, 1, 1, 2, 3, 5, 5, 8 };
vector< int > ivec( ia, ia+8 ), vres;
// ...

// поведение программы во время выполнения не определено
```

ли вы, в чем она заключается?

```
unique_copy( ivec.begin(), ivec.end(), vres.begin() );
```

Проблема вызвана тем, что алгоритм unique_copy() использует присваивание для копирования значения каждого элемента из вектора ivec, но эта операция завершится неудачно, поскольку в vres не выделено место для хранения девяти целых чисел.

Можно было бы написать две версии алгоритма unique_copy(): одна присваивает элементы, а вторая вставляет их. Эта последняя версия должна, в таком случае, поддерживать вставку в начало, в конец или в произвольное место контейнера.

Альтернативный подход, принятый в стандартной библиотеке, заключается в определении трех адаптеров, которые возвращают специальные итераторы вставки:

- back_inserter() вызывает определенную для контейнера операцию вставки push_back() вместо оператора присваивания. Аргументом back_inserter()

```
// правильно: теперь unique_copy() вставляет элементы с помощью
// vres.push_back()...
unique_copy( ivec.begin(), ivec.end(),
```

  является сам контейнер. Например, вызов unique_copy() можно исправить, написав:

```
            back_inserter( vres ) );
```

- front_inserter() вызывает определенную для контейнера операцию вставки push_front() вместо оператора присваивания. Аргументом front_inserter() тоже является сам контейнер. Заметьте, однако, что класс vector не поддерживает

```
// увы, ошибка:
    // класс vector не поддерживает операцию push_front()
    // следует использовать контейнеры deque или list
    unique_copy( ivec.begin(), ivec.end(),
```

  push_front(), так что использовать такой адаптер для вектора нельзя:

```
            front_inserter( vres ) );
```

- inserter() вызывает определенную для контейнера операцию вставки insert() вместо оператора присваивания. inserter() принимает два аргумента: сам

```
unique_copy( ivec.begin(), ivec.end(),
```

  контейнер и итератор, указывающий позицию, с которой должна начаться вставка:

```
            inserter( vres ), vres.begin() );
```

- Итератор, указывающий на позицию начала вставки, сдвигается вперед после каждой вставки, так что элементы располагаются в нужном порядке, как если бы мы

```
vector< int >::iterator iter = vres.begin(),
              iter2 = ivec.begin();

for ( ; iter2 != ivec.end() ++ iter, ++iter2 )
```

  написали:

```
    vres.insert( iter, *iter2 );
```

12.4.2. Обратные итераторы

Операции begin() и end() возвращают соответственно итераторы, указывающие на первый элемент и на элемент, расположенный за последним. Можно также вернуть обратный итератор, обходящий контейнер от последнего элемента к первому. Во всех контейнерах для поддержки такой возможности используются операции rbegin() и

```
vector< int > vec0;
const vector< int > vec1;

vector< int >::reverse_iterator r_iter0 = vec0.rbegin();
```

rend(). Есть константные и неконстантные версии обратных итераторов:

```
vector< int >::const_reverse_iterator r_iter1 = vec1.rbegin();
```

Обратный итератор применяется так же, как прямой. Разница состоит в реализации операторов перехода к следующему и предыдущему элементам. Для прямого итератора оператор ++ дает доступ к следующему элементу контейнера, тогда как для обратного – к

```
// обратный итератор обходит вектор от конца к началу
vector< type >::reverse_iterator r_iter;
for ( r_iter = vec0.rbegin();   // r_iter указывает на последний элемент
      r_iter != vec0.rend();    // пока не достигли элемента перед первым
      r_iter++ )                // переходим к предыдущему элементу
```

предыдущему. Например, для обхода вектора в обратном направлении следует написать:

```
{ /* ... */ }
```

Инвертирование семантики операторов инкремента и декремента может внести путаницу, но зато позволяет программисту передавать алгоритму пару обратных итераторов вместо прямых. Так, для сортировки вектора в порядке убывания мы передаем алгоритму

```
// сортирует вектор в порядке возрастания
sort( vec0.begin(), vec0.end() );

// сортирует вектор в порядке убывания
```

sort() пару обратных итераторов:

```
sort( vec0.rbegin(), vec0.rend() );
```

12.4.3. Потоковые итераторы

Стандартная библиотека предоставляет средства для работы потоковых итераторов чтения и записи совместно со стандартными контейнерами и обобщенными алгоритмами. Класс istream_iterator поддерживает итераторные операции с классом istream или одним из производных от него, например ifstream для работы с потоком ввода из файла. Аналогично ostream_iterator поддерживает итераторные операции с классом ostream или одним из производных от него, например ofstream для работы с потоком вывода в файл. Для использования любого из этих итераторов следует включить заголовочный файл

```
#include <iterator>
```

В следующей программе мы пользуемся потоковым итератором чтения для получения из стандартного ввода последовательности целых чисел в вектор, а затем применяем потоковый итератор записи в качестве целевого в обобщенном алгоритме

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
#include <functional>

/*
 * вход:
 * 23 109 45 89 6 34 12 90 34 23 56 23 8 89 23
 *
 * выход:
 * 109 90 89 56 45 34 23 12 8 6
 */

int main()
{
        istream_iterator< int > input( cin );
        istream_iterator< int > end_of_stream;

        vector<int> vec;
        copy ( input, end_of_stream, inserter( vec, vec.begin() ));

        sort( vec.begin(), vec.end(), greater<int>() );

        ostream_iterator< int > output( cout, " " );
        unique_copy( vec.begin(), vec.end(), output );
```

unique_copy():

```
}
```

12.4.4. Итератор istream_iterator

В общем виде объявление потокового итератора чтения `istream_iterator` имеет форму:

```
istream_iterator<Type> identifier( istream& );1.
```

Примечание [O.A.3]: Нумерация сносок сбита.

---

1. Если имеющийся у Вас компилятор пока не поддерживает параметр шаблонов по умолчанию, то конструктору `istream_iterator` необходимо будет явно передать также и второй аргумент: тип `difference_type`, способный хранить результат вычитания двух итераторов контейнера, куда помещаются элементы. Например, в разделе 12.2 при изучении программы, которая должна транслироваться компилятором, не поддерживающим параметры шаблонов по умолчанию, мы писали:

```
typedef vector<string,allocator>::difference_type diff_type
istream_iterator< string, diff_type > input_set1( infile1 ), eos;
istream_iterator< string, diff_type > input_set2( infile2 );
```

где Type – это любой встроенный или пользовательский тип класса, для которого определен оператор ввода. Аргументом конструктора может быть объект либо класса istream, например cin, либо производного от него класса с открытым типом

```
#include <iterator>
#include <fstream>
#include <string>
#include <complex>

// прочитать последовательность объектов типа complex
// из стандартного ввода
istream_iterator< complex > is_complex( cin );

// прочитать последовательность строк из именованного файла
ifstream infile( "C++Primer" );
```

наследования – ifstream:

```
istream_iterator< string > is_string( infile );
```

При каждом применении оператора инкремента к объекту типа istream_iterator читается следующий элемент из входного потока, для чего используется оператор operator>>(). Чтобы сделать то же самое в обобщенных алгоритмах, необходимо предоставить пару итераторов, обозначающих начальную и конечную позицию в файле. Начальную позицию дает istream_iterator, инициализированный объектом istream, – такой, скажем, как is_string. Для получения конечной позиции мы

```
// конструирует итератор end_of_stream, который будет служить маркером
// конца потока в итераторной паре
istream_iterator< string > end_of_stream

vector<string> text;

// правильно: передаем пару итераторов
copy( is_string, end_of_stream,
      inserter( text, text.begin() ));
```

используем специальный конструктор по умолчанию класса istream_iterator:

```

```

12.4.5. Итератор ostream_iterator

Объявление потокового итератора записи ostream_iterator может быть представлено в двух формах:

Если бы компилятор полностью удовлетворял стандарту C++, достаточно было бы написать так:

```
istream_iterator< string > input_set1( infile1 ), eos;
istream_iterator< string > input_set2( infile2 );
```

```
ostream_iterator<Type> identifier( ostream& )
ostream_iterator<Type> identifier( ostream&, char * delimiter )
```

где Type – это любой встроенный или пользовательский тип класса, для которого определен оператор вывода (operator<<). Во второй форме delimiter – это разделитель, то есть С-строка символов, которая выводится в файл после каждого элемента. Такая строка должна заканчиваться двоичным нулем, иначе поведение программы не определено (скорее всего, она аварийно завершит выполнение). В качестве аргумента ostream может выступать объект класса ostream, например cout, либо

```
#include <iterator>
#include <fstream>
#include <string>
#include <complex>

// записать последовательность объектов типа complex
// в стандартный вывод, разделяя элементы пробелами
ostream_iterator< complex > os_complex( cin, " " );

// записать последовательность строк в именованный файл
ofstream outfile( "dictionary" );
```

производного от него класса с открытым типом наследования, скажем ofstream:

```
ostream_iterator< string > os_string( outfile, "\n" );
```

Вот простой пример чтения из стандартного ввода и копирования на стандартный вывод

```
#include <iterator>
#include <algorithm>
#include <iostream>

int main()
{
   copy( istream_iterator< int >( cin ),
         istream_iterator< int >(),
         ostream_iterator< int >( cout, " " ));
```

с помощью безымянных потоковых итераторов и обобщенного алгоритма copy():

```
}
```

Ниже приведена небольшая программа, которая открывает указанный пользователем файл и копирует его на стандартный вывод, применяя для этого алгоритм copy() и потоковый итератор записи ostream_iterator:

```
#include <string>
#include <algorithm>
#include <fstream>
#include <iterator>

main()
{
          string file_name;

          cout << "please enter a file to open: ";
          cin >> file_name;

          if ( file_name.empty() || !cin ) {
          cerr << "unable to read file name\n"; return -1;
          }

          ifstream infile( file_name.c_str());
          if ( !infile ) {
          cerr << "unable to open " << file_name << endl;
          return -2;
          }

          istream_iterator< string > ins( infile ), eos;
          ostream_iterator< string > outs( cout, " " );
          copy( ins, eos, outs );

}
```

12.4.6. Пять категорий итераторов

Для поддержки полного набора обобщенных алгоритмов стандартная библиотека определяет пять категорий итераторов, положив в основу классификации множество операций. Это итераторы чтения (`InputIterator`), записи (`OutputIterator`), однонаправленные (`ForwardIterator`) и двунаправленные итераторы (`BidirectionalIterator`), а также итераторы с произвольным доступом (`RandomAccessIterators`). Ниже приводится краткое обсуждение характеристик каждой категории:

- итератор чтения можно использовать для получения элементов из контейнера, но поддержка записи в контейнер не гарантируется. Такой итератор должен обеспечивать следующие операции (итераторы, поддерживающие также дополнительные операции, можно употреблять в качестве итераторов чтения при условии, что они удовлетворяют минимальным требованиям): сравнение двух итераторов на равенство и неравенство, префиксная и постфиксная форма инкремента итератора для адресации следующего элемента (оператор `++`), чтение элемента с помощью оператора разыменования (`*`). Такого уровня поддержки требуют, в частности, алгоритмы `find()`, `accumulate()` и `equal()`. Любому алгоритму, которому необходим итератор чтения, можно передавать также и итераторы категорий, описанных в пунктах 3, 4 и 5;
- итератор записи можно представлять себе как противоположный по функциональности итератору чтения. Иными словами, его можно использовать для записи элементов контейнера, но поддержка чтения из контейнера не гарантируется. Такие итераторы обычно применяются в качестве третьего аргумента алгоритма (например, `copy()`) и указывают на позицию, с которой надо начинать копировать.

Любому алгоритму, которому необходим итератор записи, можно передавать также и итераторы других категорий, перечисленных в пунктах 3, 4 и 5;

- однонаправленный итератор можно использовать для чтения и записи в контейнер, но только в одном направлении обхода (обход в обоих направлениях поддерживается итераторами следующей категории). К числу обобщенных алгоритмов, требующих как минимум однонаправленного итератора, относятся `adjacent_find()`, `swap_range()` и `replace()`. Конечно, любому алгоритму, которому необходим подобный итератор, можно передавать также и итераторы описанных ниже категорий;
- двунаправленный итератор может читать и записывать в контейнер, а также перемещаться по нему в обоих направлениях. Среди обобщенных алгоритмов, требующих как минимум двунаправленного итератора, выделяются `place_merge()`, `next_permutation()` и `reverse()`;
- итератор с произвольным доступом, помимо всей функциональности, поддерживаемой двунаправленным итератором, обеспечивает доступ к любой позиции внутри контейнера за постоянное время. Подобные итераторы требуются таким обобщенным алгоритмам, как `binary_search()`, `sort_heap()` и `nth-element()`.

Упражнение 12.6

Объясните, почему некорректны следующие примеры. Какие ошибки обнаруживаются во

```
(a) const vector<string> file_names( sa, sa+6 );
    vector<string>::iterator it = file_names.begin()+2;

(b) const vector<int> ivec;
    fill( ivec.begin(), ivec.end(), ival );

(c) sort( ivec.begin(), ivec.end() );

(d) list<int> ilist( ia, ia+6 );
    binary_search( ilist.begin(), ilist.end() );
```

время компиляции?

```
(e) sort( ivec1.begin(), ivec3.end() );
```

Упражнение 12.7

Напишите программу, которая читает последовательность целых чисел из стандартного ввода с помощью потокового итератора чтения `istream_iterator`. Нечетные числа поместите в один файл посредством `ostream_iterator`, разделяя значения пробелом. Четные числа таким же образом запишите в другой файл, при этом каждое значение должно размещаться в отдельной строке.

12.5. Обобщенные алгоритмы

Первые два аргумента любого обобщенного алгоритма (разумеется, есть исключения, которые только подтверждают правило) – это пара итераторов, обычно называемых `first` и `last`, ограничивающих диапазон элементов внутри контейнера или встроенного массива, к которым применяется этот алгоритм. Как правило, диапазон элементов

(иногда его называют интервалом с включенной левой границей) обозначается

```
// читается так: включает первый и все последующие элементы,
// кроме последнего
```

следующим образом:

```
[ first, last )
```

Эта запись говорит о том, что диапазон начинается с элемента `first` и продолжается до элемента `last`, исключая последний. Если

```
first == last
```

то говорят, что диапазон пуст.

К паре итераторов предъявляется следующее требование: если начать с элемента `first` и последовательно применять оператор инкремента, то возможно достичь элемента `last`. Однако компилятор не в состоянии проверить выполнение этого ограничения; если оно нарушается, поведение программы не определено, обычно все заканчивается аварийным остановом и дампом памяти.

В объявлении каждого алгоритма указывается минимально необходимая категория итератора (см. раздел 12.4). Например, для алгоритма `find()`, реализующего однопроходный обход контейнера с доступом только для чтения, требуется итератор чтения, но можно передать и однонаправленный или двунаправленный итератор, а также итератор с произвольным доступом. Однако передача итератора записи приведет к ошибке. Не гарантируется, что ошибки, связанные с передачей итератора не той категории, будут обнаружены во время компиляции, поскольку категории итераторов – это не собственно типы, а лишь параметры-типы, передаваемые шаблону функции.

Некоторые алгоритмы существуют в нескольких версиях: в одной используется встроенный оператор, а во второй – объект-функция или указатель на функцию, которая предоставляет альтернативную реализацию оператора. Например, `unique()` по умолчанию сравнивает два соседних элемента с помощью оператора равенства, определенного для типа объектов в контейнере. Но если такой оператор равенства не определен или мы хотим сравнивать элементы иным способом, то можно передать либо объект-функцию, либо указатель на функцию, обеспечивающую нужную семантику. Встречаются также алгоритмы с похожими, но разными именами. Так, предикатные версии всегда имеют имя, оканчивающееся на `_if`, например `find_if()`. Скажем, есть алгоритм `replace()`, реализованный с помощью встроенного оператора равенства, и `replace_if()`, которому передается объект-предикат или указатель на функцию.

Алгоритмы, модифицирующие контейнер, к которому они применяются, обычно имеют две версии: одна преобразует содержимое контейнера по месту, а вторая возвращает копию исходного контейнера, в которой и отражены все изменения. Например, есть алгоритмы `replace()` и `replace_copy()` (имя версии с копированием всегда заканчивается на `_copy`). Однако не у всех алгоритмов, модифицирующих контейнер, имеется такая версия. К примеру, ее нет у алгоритма `sort()`. Если же мы хотим, чтобы сортировалась копия, то создать и передать ее придется самостоятельно.

Для использования любого обобщенного алгоритма необходимо включить в программу заголовочный файл

```
#include <algorithm>
```

А для любого из четырех численных алгоритмов – adjacent_differences(), accumulate(), inner_product() и partial_sum() – включить также заголовок

```
#include <numeric>
```

Все существующие алгоритмы для удобства изложения распределены нами на девять категорий (они перечислены ниже). В Приложении алгоритмы рассматриваются в алфавитном порядке, и для каждого приводится пример применения.

12.5.1. Алгоритмы поиска

Тринадцать алгоритмов поиска предоставляют различные способы нахождения определенного значения в контейнере. Три алгоритма equal_range(), lower_bound() и upper_bound() выполняют ту или иную форму двоичного поиска. Они показывают, в

```
adjacent_find(), binary_search(), count(),count_if(), equal_range(),
find(), find_end(), find_first_of(), find_if(), lower_bound(),
```

какое место контейнера можно вставить новое значение, не нарушая порядка сортировки.

```
upper_bound(), search(), search_n()
```

12.5.2. Алгоритмы сортировки и упорядочения

Четырнадцать алгоритмов сортировки и упорядочения предлагают различные способы упорядочения элементов контейнера. Разбиение (partition) – это разделение элементов контейнера на две группы: удовлетворяющие и не удовлетворяющие некоторому условию. Так, можно разбить контейнер по признаку четности/нечетности чисел или в зависимости от того, начинается слово с заглавной или со строчной буквы. Устойчивый (stable) алгоритм сохраняет относительный порядок элементов с одинаковыми значениями или удовлетворяющих одному и тому же условию. Например, если дана последовательность:

```
{ "pshew", "honey", "Tigger", "Pooh" }
```

то устойчивое разбиение по наличию/отсутствию заглавной буквы в начале слова генерирует последовательность, в которой относительный порядок слов в каждой категории сохранен:

```
{ "Tigger", "Pooh", "pshew", "honey" }
```

При использовании неустойчивой версии алгоритма сохранение порядка не гарантируется. (Отметим, что алгоритмы сортировки нельзя применять к списку и

```
inplace_merge(), merge(), nth_element(), partial_sort(),
partial_sort_copy(), partition(), random_shuffle(), reverse(),
reverse_copy(), rotate(), rotate_copy(), sort(), stable_sort(),
```

ассоциативным контейнерам, таким, как множество (set) или отображение (map).)

```
stable_partition()
```

12.5.3. Алгоритмы удаления и подстановки

Пятнадцать алгоритмов удаления и подстановки предоставляют различные способы замены или исключения одного элемента или целого диапазона. unique() удаляет одинаковые соседние элементы. iter_swap() обменивает значения элементов,

```
copy(), copy_backwards(), iter_swap(), remove(), remove_copy(),
remove_if(),remove_if_copy(), replace(), replace_copy(),
replace_if(), replace_copy_if(), swap(), swap_range(), unique(),
```

адресованных парой итераторов, но не модифицирует сами итераторы.

```
unique_copy()
```

12.5.4. Алгоритмы перестановки

Рассмотрим последовательность из трех символов: {a,b,c}. Для нее существует шесть различных перестановок: abc, acb, bac, bca, cab и cba, лексикографически упорядоченных на основе оператора "меньше". Таким образом, abc – это первая перестановка, потому что каждый элемент меньше последующего. Следующая перестановка – acb, поскольку в начале все еще находится a – наименьший элемент последовательности. Соответственно перестановки, начинающиеся с b, предшествуют тем, которые начинаются с c. Из bac и bca меньшей является bac, так как последовательность ac лексикографически меньше, чем ca. Если дана перестановка bca, то можно сказать, что предшествующей для нее будет bac, а последующей – cab. Для перестановки abc нет предшествующей, а для cba – последующей.

```
next_permutation(), prev_permutation()
```

12.5.5. Численные алгоритмы

Следующие четыре алгоритма реализуют численные операции с контейнером. Для их использования необходимо включить заголовочный файл <numeric>.

```
accumulate(), partial_sum(), inner_product(), adjacent_difference()
```

12.5.6. Алгоритмы генерирования и модификации

Шесть алгоритмов генерирования и модификации либо создают и заполняют новую последовательность, либо изменяют значения в существующей.

```
fill(), fill_n(), for_each(), generate(),generate_n(), transform()
```

12.5.7. Алгоритмы сравнения

Семь алгоритмов дают разные способы сравнения одного контейнера с другим (алгоритмы min() и max() сравнивают два элемента). Алгоритм

`lexicographical_compare()` выполняет лексикографическое (словарное) упорядочение

```
equal(), includes(), lexicographical_compare(), max(), max_element(),
```

(см. также обсуждение перестановок и Приложение).

```
min(), min_element(), mismatch()
```

12.5.8. Алгоритмы работы с множествами

Четыре алгоритма этой категории реализуют теоретико-множественные операции над любым контейнерным типом. При объединении создается отсортированная последовательность элементов, принадлежащих хотя бы одному контейнеру, при пересечении – обоим контейнерам, а при взятии разности – принадлежащих первому контейнеру, но не принадлежащих второму. Наконец, симметрическая разность – это отсортированная последовательность элементов, принадлежащих одному из контейнеров,

```
set_union(), set_intersection(), set_difference(),
```

но не обоим.

```
set_symmetric_difference()
```

12.5.9. Алгоритмы работы с хипом

Хип (heap) – это разновидность двоичного дерева, представленного в массиве. Стандартная библиотека предоставляет такую реализацию хипа, в которой значение ключа в любом узле больше либо равно значению ключа в любом потомке этого узла.

```
make_heap(), pop_heap(), push_heap(), sort_heap()
```

12.6. Когда нельзя использовать обобщенные алгоритмы

Ассоциативные контейнеры (отображения и множества) поддерживают определенный порядок элементов для быстрого поиска и извлечения. Поэтому к ним не разрешается применять обобщенные алгоритмы, меняющие порядок, такие, как `sort()` и `partition()`. Если в ассоциативном контейнере требуется переставить элементы, то необходимо сначала скопировать их в последовательный контейнер, например в вектор или список.

Контейнер `list` (список) реализован в виде двусвязного списка: в каждом элементе, помимо собственно данных, хранятся два члена-указателя – на следующий и на предыдущий элементы. Основное преимущество списка – это эффективная вставка и удаление одного элемента или целого диапазона в произвольное место списка, а недостаток – невозможность произвольного доступа. Например, можно написать:

```
vector<string>::iterator vec_iter = vec.begin() + 7;
```

Такая форма вполне допустима и инициализирует vec_iter адресом восьмого элемента

```
// ошибка: арифметические операции над итераторами
// не поддерживаются списком
```

вектора, но запись

```
list<string>::iterator list_iter = slist.begin() + 7;
```

некорректна, так как элементы списка не занимают непрерывную область памяти. Для того чтобы добраться до восьмого элемента, необходимо посетить все промежуточные.

Поскольку список не поддерживает произвольного доступа, то алгоритмы merge(), remove(), reverse(), sort() и unique() лучше к таким контейнерам не применять, хотя ни один из них явно не требует наличия соответствующего итератора. Вместо этого для списка определены специализированные версии названных операций в виде функций-членов, а также операция splice():

- list::merge() объединяет два отсортированных списка
- list::remove() удаляет элементы с заданным значением
- list::remove_if()удаляет элементы, удовлетворяющие некоторому условию
- list::reverse() переставляет элементы списка в обратном порядке
- list::sort() сортирует элементы списка
- list::splice() перемещает элементы из одного списка в другой
- list::unique() оставляет один элемент из каждой цепочки одинаковых смежных элементов

```
void list::merge( list rhs );
template <class Compare>
```

12.6.1. Операция list_merge()

```
   void list::merge( list rhs, Compare comp );
```

Элементы двух упорядоченных списков объединяются либо на основе оператора “меньше”, определенного для типа элементов в контейнере, либо на основе указанной пользователем операции сравнения. (Заметьте, что элементы списка rhs *перемещаются* в список, для которого вызвана функция-член merge(); по завершении операции список rhs будет пуст.) Например:

```
int array1[ 10 ] = { 34, 0, 8, 3, 1, 13, 2, 5, 21, 1 };
int array2[ 5 ] = { 377, 89, 233, 55, 144 };

list< int > ilist1( array1, array1 + 10 );
list< int > ilist2( array2, array2 + 5 );

// для объединения требуется, чтобы оба списка были упорядочены
ilist1.sort(); ilist2.sort();

ilist1.merge( ilist2 );
```

После выполнения операции `merge()` список `ilist2` пуст, а `ilist1` содержит первые 15 чисел Фибоначчи в порядке возрастания.

12.6.2. Операция list::remove()

```
void list::remove( const elemType &value );
```

Операция `remove()` удаляет все элементы с заданным значением:

```
ilist1.remove( 1 );
```

```
template < class Predicate >
```

12.6.3. Операция list::remove_if()

```
   void list::remove_if( Predicate pred );
```

Операция `remove_if()` удаляет все элементы, для которых выполняется указанное

```
class Even {
public:
   bool operator()( int elem ) { return ! (elem % 2 ); }
};
```

условие, т.е. предикат `pred` возвращает `true`. Например:

```
ilist1.remove_if( Even() );
```

удаляет все четные числа из списка, определенного при рассмотрении `merge()`.

12.6.4. Операция list::reverse()

```
void list::reverse();
```

Операция reverse() изменяет порядок следования элементов списка на противоположный:

```
ilist1.reverse();
```

```
void list::sort();
template <class Compare>
```

12.6.5. Операция list::sort()

```
   void list::sort( Compare comp );
```

По умолчанию sort() упорядочивает элементы списка по возрастанию с помощью оператора “меньше”, определенного в классе элементов контейнера. Вместо этого можно явно передать в качестве аргумента оператор сравнения. Так,

```
list1.sort();
```

упорядочивает list1 по возрастанию, а

```
list1.sort( greater<int>() );
```

упорядочивает list1 по убыванию, используя оператор “больше”.

```
void list::splice( iterator pos, list rhs );
void list::splice( iterator pos, list rhs, iterator ix );
void list::splice( iterator pos, list rhs,
```

12.6.6. Операция list::splice()

```
                  iterator first, iterator last );
```

Операция splice() имеет три формы: перемещение одного элемента, всех элементов или диапазона из одного списка в другой. В каждом случае передается итератор, указывающий на позицию вставки, а перемещаемые элементы располагаются

```
int array[ 10 ] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
list< int > ilist1( array, array + 10 );
```

непосредственно перед ней. Если даны два списка:

```
list< int > ilist2( array, array + 2 );       // содержит 0, 1
```

то следующее обращение к splice() перемещает первый элемент ilist1 в ilist2. Теперь ilist2 содержит элементы 0, 1 и 0, тогда как в ilist1 элемента 0 больше нет.

```
// ilist2.end() указывает на позицию, куда нужно переместить элемент
// элементы вставляются перед этой позицией
// ilist1 указывает на список, из которого перемещается элемент
// ilist1.begin() указывает на сам перемещаемый элемент

ilis2.splice( ilist2.end(), ilist1, ilist1.begin() );
```

В следующем примере применения splice() передаются два итератора,

```
list< int >::iterator first, last;

first = ilist1.find( 2 );
last  = ilist1.find( 13 );
```

ограничивающие диапазон перемещаемых элементов:

```
ilist2.splice( ilist2.begin(), ilist1, first, last );
```

В данном случае элементы 2, 3, 5 и 8 удаляются из ilist1 и вставляются в начало ilist2. Теперь ilist1 содержит пять элементов 1, 1, 13, 21 и 34. Для их перемещения

```
list< int >::iterator pos = ilist2.find( 5 );
```

в ilist2 можно воспользоваться третьей вариацией операции splice():

```
ilist2.splice( pos, ilist1 );
```

Итак, список ilist1 пуст. Последние пять элементов перемещены в позицию списка ilist2, предшествующую той, которую занимает элемент 5.

```
void list::unique();
template <class BinaryPredicate>
```

12.6.7. Операция list::unique()

```
   void list::unique( BinaryPredicate pred );
```

Операция unique() удаляет соседние дубликаты. По умолчанию при сравнении используется оператор равенства, определенный для типа элементов контейнера. Например, если даны значения {0,2,4,6,4,2,0}, то после применения unique() список останется таким же, поскольку в соседних позициях дубликатов нет. Но если мы сначала отсортируем список, что даст {0,0,2,2,4,4,6}, а потом применим unique(), то получим четыре различных значения {0,2,4,6}.

```
ilist.unique();
```

Вторая форма unique() принимает альтернативный оператор сравнения. Например,

```
class EvenPair {
public:
   bool operator()( int val1, val2 )
        { return ! (val2 % val1 );   }
};
```

```
ilist.unique( EvenPair() );
```

удаляет соседние элементы, если второй элемент без остатка делится на первый.

Эти операции, являющиеся членами класса, следует предпочесть соответствующим обобщенным алгоритмам при работе со списками. Остальные обобщенные алгоритмы, такие, как `find()`, `transform()`, `for_each()` и т.д., работают со списками так же эффективно, как и с другими контейнерами (еще раз напомним, что подробно все алгоритмы рассматриваются в Приложении).

Упражнение 12.8

Измените программу из раздела 12.2, используя список вместо вектора.

Часть IV
Объектное программирование

В части 4 мы сосредоточимся на объектном программировании, т.е. на применении классов C++ для определения новых типов, манипулировать которыми так же просто, как и встроенными. Создавая новые типы для описания предметной области, C++ помогает программисту писать более легкие для понимания приложения. Классы позволяют отделить детали, касающиеся реализации нового типа, от определения интерфейса и операций, предоставляемых пользователю. При этом уделяется меньше внимания мелочам, из-за чего программирование становится таким утомительным занятием. Значимые для приложения типы можно реализовать всего один раз, после чего использовать повторно. Средства, обеспечивающие инкапсуляцию данных и функций, необходимых для реализации типа, помогают значительно упростить последующее сопровождение и развитие приложения.

В главе 13 мы рассмотрим общий механизм классов: порядок их определения, концепцию *сокрытия информации* (т.е. отделение открытого интерфейса от закрытой реализации), способы определения и манипулирования объектами класса, область видимости, вложенные классы и классы как члены пространства имен.

В главе 14 изучаются предоставляемые C++ средства инициализации и уничтожения объектов класса, а также присваивания им значений путем применения таких специальных функций-членов класса, как *конструкторы*, *деструкторы* и *копирующие конструкторы*. Мы рассмотрим вопрос о почленной инициализации и копировании, когда объект класса инициализируется или ему присваивается значение другого объекта того же класса.

В главе 15 мы расскажем о перегрузке операторов, которая позволяет использовать операнды типа класса со встроенными операторами, описанными в главе 4. Таким образом, работа с объектами типа класса может быть сделана столь же понятной, как и работа со встроенными типами. В начале главы 15 представлены общие концепции и соображения, касающиеся проектирования перегрузки операторов, а затем рассмотрены конкретные операторы, такие, как присваивание, взятие индекса, вызов, а также специфичные для классов операторы `new` и `delete`. Иногда необходимо объявить перегруженный оператор, как друга класса, наделив его специальными правами доступа, в данной главе объясняется, зачем это нужно. Здесь же представлен еще один специальный вид функций-членов – *конвертеры*, которые позволяют программисту определить стандартные преобразования. Конвертеры неявно применяются компилятором, когда объекты класса используются в качестве фактических аргументов функции или операндов встроенного либо перегруженного оператора. Завершается глава изложением правил разрешения перегрузки функций с учетом аргументов типа класса, функций-членов и перегруженных операторов.

Тема главы 16 – шаблоны классов. Шаблон – это предписание для создания класса, в котором один или несколько типов параметризованы. Например, `vector` может быть параметризован типом элементов, хранящихся в нем, а `buffer` – типом элементов в буфере или его размером. В этой главе объясняется, как определить и конкретизировать шаблон. Поддержка классов в C++ теперь рассматривается иначе – в свете наличия шаблонов, и снова обсуждаются функции-члены, объявления друзей и вложенные типы. Здесь мы еще раз вернемся к модели компиляции шаблонов, описанной в главе 10, чтобы показать, какое влияние оказывают на нее шаблоны классов.

13

13. Классы

Механизм классов в C++ позволяет пользователям определять собственные типы данных. По этой причине их часто называют пользовательскими типами. Класс может наделять дополнительной функциональностью уже существующий тип. Так, например, IntArray, введенный в главе 2, предоставляет больше возможностей, чем тип “массив int”. С помощью классов можно создавать абсолютно новые типы, например `Screen` (экран) или `Account` (расчетный счет). Как правило, классы используются для абстракций, не отражаемых встроенными типами адекватно.

В этой главе мы узнаем, как определять типы и использовать объекты классов; увидим, что определение класса вводит как данные-члены, описывающие его, так и функции-члены, составляющие набор операций, применимых к объектам класса. Мы покажем, как можно обеспечить сокрытие информации, объявив внутреннее представление и реализацию закрытыми, но открыв операции над объектами. Говорят, что закрытое внутреннее представление инкапсулировано, а открытую часть класса называют его интерфейсом.

Далее в этой главе мы познакомимся с особым видом членов класса – статическими членами. Мы расскажем также, как можно использовать указатели на члены и функции-члены класса, и рассмотрим объединения, представляющие собой специализированный вид класса для хранения объектов разных типов в одной области памяти. Завершается глава обсуждением области видимости класса и описанием правил разрешения имен в этой области; затрагиваются такие понятия, как вложенные классы, классы-члены пространства имен и локальные классы.

13.1. Определение класса

Определение класса состоит из двух частей: *заголовка*, включающего ключевое слово `class`, за которым следует имя класса, и *тела*, заключенного в фигурные скобки. После

```
class Screen { /* ... */ };
```

такого определения должны стоять точка с запятой или список объявлений:

```
class Screen { /* ... */ } myScreen, yourScreen;
```

Внутри тела объявляются данные-члены и функции-члены и указываются уровни доступа к ним. Таким образом, тело класса определяет *список его членов*.

Каждое определение вводит новый тип данных. Даже если два класса имеют одинаковые списки членов, они все равно считаются разными типами:

```
class First {
   int memi;
   double memd;
};

class Second {
   int memi;
   double memd;
};

class First obj1;

Second obj2 = obj1;   // ошибка: obj1 и obj2 имеют разные типы
```

Тело класса определяет отдельную область видимости. Объявление членов внутри тела помещает их имена в область видимости класса. Наличие в двух разных классах членов с одинаковыми именами – не ошибка, эти имена относятся к разным объектам. (Подробнее об областях видимости классов мы поговорим в разделе 13.9.)

После того как тип класса определен, на него можно ссылаться двумя способами:

- написать ключевое слово `class`, а после него – имя класса. В предыдущем примере объект `obj1` класса `First` объявлен именно таким образом;
- указать только имя класса. Так объявлен объект `obj2` класса `Second` из приведенного примера.

Оба способа сослаться на тип класса эквивалентны. Первый заимствован из языка C и остается корректным методом задания типа класса; второй способ введен в C++ для упрощения объявлений.

13.1.1. Данные-члены

Данные-члены класса объявляются так же, как переменные. Например, у класса `Screen`

```
#include <string>
class Screen {
   string             _screen;   // string( _height * _width )
   string::size_type  _cursor;   // текущее положение на экране
   short              _height;   // число строк
   short              _width;    // число колонок
};
```

могут быть следующие данные-члены:

Поскольку мы решили использовать строки для внутреннего представления объекта класса `Screen`, то член `_screen` имеет тип `string`. Член `_cursor` – это смещение в строке, он применяется для указания текущей позиции на экране. Для него использован переносимый тип `string::size_type`. (Тип `size_type` рассматривался в разделе 6.8.)

Необязательно объявлять два члена типа `short` по отдельности. Вот объявление класса `Screen`, эквивалентное приведенному выше:

```
class Screen {
/*
 * _ screen адресует строку размером _height * _width
 * _cursor указывает текущую позицию на экране
 * _height и _width - соответственно число строк и колонок
 */
   string             _screen;
   string::size_type  _cursor;
   short              _height, _width;
};
```

```
class StackScreen {
   int topStack;
   void (*handler)();     // указатель на функцию
   vector<Screen> stack;  // вектор классов
};
```

Член класса может иметь любой тип:

```
class First {
   int    memi = 0;    // ошибка
   double memd = 0.0;  // ошибка
};
```

Описанные данные-члены называются *нестатическими*. Класс может иметь также и *статические* данные-члены. (У них есть особые свойства, которые мы рассмотрим в разделе 13.5.)

Объявления данных-членов очень похожи на объявления переменных в области видимости блока или пространства имен. Однако их, за исключением статических членов, нельзя явно инициализировать в теле класса:

```
class Screen {
/*
 * _ screen адресует строку размером _height * _width
 * _cursor указывает текущую позицию на экране
 * _height и _width - соответственно число строк и колонок
 */
   string             _screen;
   string::size_type  _cursor;
   short              _height, _width;
};
```

Данные-члены класса инициализируются с помощью конструктора класса. (Мы рассказывали о конструкторах в разделе 2.3; более подробно они рассматриваются в главе 14.)

13.1.2. Функции-члены

Пользователям, по-видимому, понадобится широкий набор операций над объектами типа `Screen`: возможность перемещать курсор, проверять и устанавливать области экрана и рассчитывать его реальные размеры во время выполнения, а также копировать один объект в другой. Все эти операции можно реализовать с помощью функций-членов.

Функции-члены класса объявляются в его теле. Это объявление выглядит точно так же, как объявление функции в области видимости пространства имен. (Напомним, что глобальная область видимости – это тоже область видимости пространства имен. Глобальные функции рассматривались в разделе 8.2, а пространства имен – в разделе 8.5.) Например:

```
class Screen {
public:
   void home();
   void move( int, int );
   char get();
   char get( int, int );
   void checkRange( int, int );
   // ...
};
```

```
class Screen {
public:
   // определения функций home() и get()
   void home() { _cursor = 0; }
   char get() { return _screen[_cursor]; }
   // ...
};
```

Определение функции-члена также можно поместить внутрь тела класса:

`home()` перемещает курсор в левый верхний угол экрана; `get()` возвращает символ, находящийся в текущей позиции курсора.

Функции-члены отличаются от обычных функций следующим:

- функция-член объявлена в области видимости своего класса, следовательно, ее имя не видно за пределами этой области. К функции-члену можно обратиться с помощью

```
ptrScreen->home();
```

  одного из операторов доступа к членам – точки (`.`) или стрелки (`->`):

```
myScreen.home();
```

  (в разделе 13.9 область видимости класса обсуждается более детально);

- функции-члены имеют право доступа как к открытым, так и к закрытым членам класса, тогда как обычным функциям доступны лишь открытые. Конечно, функции-члены одного класса, как правило, не имеют доступа к данным-членам другого класса.

Функция-член может быть перегруженной (перегруженные функции рассматриваются в главе 9). Однако она способна перегружать лишь другую функцию-член своего класса. По отношению к функциям, объявленным в других классах или пространствах имен, функция-член находится в отдельной области видимости и, следовательно, не может перегружать их. Например, объявление `get(int, int)` перегружает лишь `get()` из того же класса `Screen`:

```
class Screen {
public:
   // объявления перегруженных функций-членов get()
   char get() { return _screen[_cursor]; }
   char get( int, int );
   // ...
};
```

(Подробнее мы остановимся на функциях-членах класса в разделе 13.3.)

13.1.3. Доступ к членам

Часто бывает так, что внутреннее представление типа класса изменяется в последующих версиях программы. Допустим, опрос пользователей нашего класса Screen показал, что для его объектов всегда задается размер экрана 80 × 24. В таком случае было бы желательно заменить внутреннее представление экрана менее гибким, но более

```
class Screen {
public:
   // функции-члены
private:
   // инициализация статических членов (см. 13.5)
   static const int   _height = 24;
   static const int   _width = 80;
   string             _screen;
   string::size_type  _cursor;
};
```

эффективным:

Прежняя реализация функций-членов (то, как они манипулируют данными-членами класса) больше не годится, ее нужно переписать. Но это не означает, что должен измениться и интерфейс функций-членов (список формальных параметров и тип возвращаемого значения).

Если бы данные-члены класса Screen были открыты и доступны любой функции внутри программы, как отразилось бы на пользователях изменение внутреннего представления этого класса?

- все функции, которые напрямую обращались к данным-членам старого представления, перестали бы работать. Следовательно, пришлось бы отыскивать и изменять соответствующие части кода;
- так как интерфейс не изменился, то коды, манипулировавшие объектами класса Screen только через функции-члены, не пришлось бы модифицировать. Но поскольку сами функции-члены все же изменились, программу пришлось бы откомпилировать заново.

Сокрытие информации – это формальный механизм, предотвращающий прямой доступ к внутреннему представлению типа класса из функций программы. Ограничение доступа к членам задается с помощью секций тела класса, помеченных ключевыми словами public, private и protected – *спецификаторами доступа*. Члены, объявленные в секции public, называются открытыми, а объявленные в секциях private и protected соответственно закрытыми или защищенными.

- *открытый член* доступен из любого места программы. Класс, скрывающий информацию, оставляет открытыми только функции-члены, определяющие операции, с помощью которых внешняя программа может манипулировать его объектами;
- *закрытый член* доступен только функциям-членам и *друзьям* класса. Класс, который хочет скрыть информацию, объявляет свои данные-члены закрытыми;
- *защищенный член* ведет себя как открытый по отношению к *производному классу* и как закрытый по отношению к остальной части программы. (В главе 2 мы видели пример использования защищенных членов в классе `IntArray`. Детально они рассматриваются в главе 17, где вводится понятие *наследования*.)

```
class Screen {
public:
   void home() { _cursor = 0; }
   char get() { return _screen[_cursor]; }
   char get( int, int );
   void move( int, int );
   // ...
private:
   string            _screen;
   string::size_type  _cursor;
   short             _height, _width;
```

В следующем определении класса `Screen` указаны секции `public` и `private`:

```
};
```

Согласно принятому соглашению, сначала объявляются открытые члены класса. (Обсуждение того, почему в старых программах C++ сначала шли закрытые члены и почему этот стиль еще кое-где сохранился, см. в книге [LIPPMAN96a].) В теле класса может быть несколько секций `public`, `protected` и `private`. Каждая секция продолжается либо до метки следующей секции, либо до закрывающей фигурной скобки. Если спецификатор доступа не указан, то секция, непосредственно следующая за открывающей скобкой, по умолчанию считается `private`.

13.1.4. Друзья

Иногда удобно разрешить некоторым функциям доступ к закрытым членам класса. Механизм *друзей* позволяет классу разрешать доступ к своим неоткрытым членам.

Объявление друга начинается с ключевого слова `friend` и может встречаться только внутри определения класса. Так как друзья не являются членами класса, то не имеет значения, в какой секции они объявлены. В примере ниже мы сгруппировали все

```
class Screen {
   friend istream&
      operator>>( istream&, Screen& );
   friend ostream&
      operator<<( ostream&, const Screen& );
public:
   // ... оставшаяся часть класса Screen
```

подобные объявления сразу после заголовка класса:

```
};
```

Операторы ввода и вывода теперь могут напрямую обращаться к закрытым членам

```
#include <iostream>
ostream& operator<<( ostream& os, const Screen& s )
{
   // правильно: можно обращаться к _height, _width и _screen
   os << "<" << s._height
      << "," << s._width << ">";
   os << s._screen;

   return os;
}
```

класса Screen. Простая реализация оператора вывода выглядит следующим образом:

Другом может быть функция из пространства имен, функция-член другого класса или даже целый класс. В последнем случае всем его функциям-членам предоставляется доступ к неоткрытым членам класса, объявляющего дружественные отношения. (В разделе 15.2 друзья обсуждаются более подробно.)

13.1.5. Объявление и определение класса

О классе говорят, что он *определен*, как только встретилась скобка, закрывающая его тело. После этого становятся известными все члены класса, а следовательно, и его размер.

Можно объявить класс, не определяя его. Например:

```
class Screen;   // объявление класса Screen
```

Это объявление вводит в программу имя Screen и указывает, что оно относится к типу класса.

Тип объявленного, но еще не определенного класса допустимо использовать весьма ограниченно. Нельзя определять объект типа класса, если сам класс еще не определен, поскольку размер класса в этом момент неизвестен и компилятор не знает, сколько памяти отвести под объект.

Однако указатель или ссылку на объект такого класса объявлять можно, так как они имеют фиксированный размер, не зависящий от типа. Но, поскольку размеры класса и его членов неизвестны, применять оператор разыменования (*) к такому указателю, а также использовать указатель или ссылку для обращения к члену не разрешается, пока класс не будет полностью определен.

Член некоторого класса можно объявить принадлежащим к типу какого-либо класса только тогда, когда компилятор уже видел определение этого класса. До этого объявляются лишь члены, являющиеся указателями или ссылками на такой тип. Ниже приведено определение StackScreen, один из членов которого служит указателем на Screen, который объявлен, но еще не определен:

```
class Screen;   // объявление
class StackScreen {
   int topStack;
   // правильно: указатель на объект Screen
   Screen *stack;
   void (*handler)();
};
```

Поскольку класс не считается определенным, пока не закончилось его тело, то в нем не может быть данных-членов его собственного типа. Однако класс считается объявленным, как только распознан его заголовок, поэтому в нем допустимы члены, являющиеся

```
class LinkScreen {
   Screen window;
   LinkScreen *next;
   LinkScreen *prev;
};
```

ссылками или указателями на его тип. Например:

Упражнение 13.1

```
string _name;
```

Пусть дан класс Person со следующими двумя членами:

```
string _address;
```

```
Person( const string &n, const string &s )
      : _name( n ), _address( a ) { }
string name() { return _name; }
```

и такие функции-члены:

```
string address() { return _address; }
```

Какие члены вы объявили бы в секции public, а какие – в секции private? Поясните свой выбор.

Упражнение 13.2

Объясните разницу между объявлением и определением класса. Когда вы стали бы использовать объявление класса? А определение?

13.2. Объекты классов

Определение класса, например Screen, не приводит к выделению памяти. Память выделяется только тогда, когда определяется объект типа класса. Так, если имеется следующая реализация Screen:

```
class Screen {
public:
   // функции-члены
private:
   string           _screen;
   string:size_type _cursor;
   short            _height;
   short            _width;
};
```

то определение

```
Screen myScreen;
```

выделяет область памяти, достаточную для хранения четырех членов Screen. Имя myScreen относится к этой области. У каждого объекта класса есть собственная копия данных-членов. Изменение членов myScreen не отражается на значениях членов любого другого объекта типа Screen.

Область видимости объекта класса зависит от его положения в тексте программы. Он

```
class Screen {
   // список членов
};

int main()
{
   Screen mainScreen;
}
```

определяется в иной области, нежели сам тип класса:

Тип Screen объявлен в глобальной области видимости, тогда как объект mainScreen – в локальной области функции main().

Объект класса также имеет время жизни. В зависимости от того, где (в области видимости пространства имен или в локальной области) и как (статическим или нестатическим) он объявлен, он может существовать в течение всего времени выполнения программы или только во время вызова некоторой функции. Область видимости объекта класса и его время жизни ведут себя очень похоже. (Понятия области видимости и времени жизни введены в главе 8.)

Объекты одного и того же класса можно инициализировать и присваивать друг другу. По умолчанию копирование объекта класса эквивалентно копированию всех его членов.

```
Screen bufScreen = myScreen;
// bufScreen._height = myScreen._height;
// bufScreen._width = myScreen._width;
// bufScreen._cursor = myScreen._cursor;
```

Например:

```
// bufScreen._screen = myScreen._screen;
```

Указатели и ссылки на объекты класса также можно объявлять. Указатель на тип класса разрешается инициализировать адресом объекта того же класса или присвоить ему такой адрес. Аналогично ссылка инициализируется l-значением объекта того же класса. (В объектно-ориентированном программировании указатель или ссылка на объект базового

```
int main()
{
   Screen myScreen, bufScreen[10];
   Screen *ptr = new Screen;
   myScreen = *ptr;
   delete ptr;
   ptr = bufScreen;
   Screen &ref = *ptr;
   Screen &ref2 = bufScreen[6];
}
```

класса могут относиться и к объекту производного от него класса.)

По умолчанию объект класса передается по значению, если он выступает в роли аргумента функции или ее возвращаемого значения. Можно объявить формальный параметр функции или возвращаемое ею значение как указатель или ссылку на тип класса. (В разделе 7.3 были представлены параметры, являющиеся указателями или ссылками на типы классов, и объяснялось, когда их следует использовать. В разделе 7.4 с этой точки зрения рассматривались типы возвращаемых значений.)

Для доступа к данным или функциям-членам объекта класса следует пользоваться соответствующими операторами. Оператор “точка” (`.`) применяется, когда операндом является сам объект или ссылка на него; а “стрелка” (`->`) – когда операндом служит

```
#include "Screen.h"

bool isEqual( Screen& s1, Screen *s2 )
{ // возвращает false, если объекты не равны, и true - если равны

   if (s1.height() != s2->height() ||
       s2.width() != s2->width() )
          return false;

   for ( int ix = 0; ix < s1.height(); ++ix )
      for ( int jy = 0; jy < s2->width(); ++jy )
         if ( s1.get( ix, jy ) != s2->get( ix, jy ) )
            return false;

   return true;    // попали сюда? значит, объекты равны
}
```

указатель на объект:

`isEqual()` – это не являющаяся членом функция, которая сравнивает два объекта `Screen`. У нее нет права доступа к закрытым членам `Screen`, поэтому напрямую обращаться к ним она не может. Сравнение проводится с помощью открытых функций-членов данного класса.

Для получения высоты и ширины экрана `isEqual()` должна пользоваться функциями-членами `height()` и `width()` для чтения закрытых членов класса. Их реализация тривиальна:

```
class Screen {
public:
   int height() { return _height; }
   int width()  { return _width; }
   // ...
private:
   short _heigh, _width;
   // ...
};
```

Применение оператора доступа к указателю на объект класса эквивалентно последовательному выполнению двух операций: применению оператора разыменования (*) к указателю, чтобы получить адресуемый объект, и последующему применению оператора “точка” для доступа к нужному члену класса. Например, выражение

```
s2->height()
```

можно переписать так:

```
(*s2).height()
```

Результат будет одним и тем же.

13.3. Функции-члены класса

Функции-члены реализуют набор операций, применимых к объектам класса. Например,

```
class Screen {
public:
   void home() { _cursor = 0; }
   char get() { return _screen[_cursor]; }
   char get( int, int );
   void move( int, int );
   bool checkRange( int, int );
   int height() { return _height; }
   int width()  { return _width; }
   // ...
};
```

для Screen такой набор состоит из следующих объявленных в нем функций-членов:

```
Screen myScreen, groupScreen;
myScreen.home();
```

Хотя у любого объекта класса есть собственная копия всех данных-членов, каждая функция-член существует в единственном экземпляре:

```
groupScreen.home();
```

При вызове функции home() для объекта myScreen происходит обращение к его члену _cursor. Когда же эта функция вызывается для объекта groupScreen, то она обращается

к члену _cursor именно этого объекта, причем сама функция home() одна и та же. Как же может одна функция-член обращаться к данным-членам разных объектов? Для этого применяется указатель this, рассматриваемый в следующем разделе.

13.3.1. Когда использовать встроенные функции-члены

Обратите внимание, что определения функций home(), get(), height() и width() приведены прямо в теле класса. Такие функции называются *встроенными*. (Мы говорили об этом в разделе 7.6.)

Функции-члены можно объявить в теле класса встроенными и явно, поместив перед

```
class Screen {
public:
   // использование ключевого слова inline
   // для объявления встроенных функций-членов
   inline void home() { _cursor = 0; }
   inline char get() { return _screen[_cursor]; }
   // ...
```

типом возвращаемого значения ключевое слово inline:

```
};
```

Определения home() и get() в приведенных примерах эквивалентны. Поскольку ключевое слово inline избыточно, мы в этой книге не пишем его явно для функций-членов, определенных в теле класса.

Функции-члены, состоящие из двух или более строк, лучше определять вне тела. Для идентификации функции как члена некоторого класса требуется специальный синтаксис объявления: имя функции должно быть *квалифицировано* именем ее класса. Вот как

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

// имя функции-члена квалифицировано именем Screen::
bool Screen::checkRange( int row, int col )
{ // проверить корректность координат
   if ( row < 1 || row > _height ||
        col < 1 || col > _width ) {
      cerr << "Screen coordinates ( "
           << row << ", " << col
           << " ) out of bounds.\n";
      return false;
    }
    return true;
```

выглядит определение функции checkRange(), квалифицированное именем Screen:

```
}
```

Прежде чем определять функцию-член вне тела класса, необходимо объявить ее внутри тела, обеспечив ее видимость. Например, если бы перед определением функции checkRange() не был включен заголовочный файл Screen.h, то компилятор выдал бы сообщение об ошибке. Тело класса определяет полный список его членов. Этот список не может быть расширен после закрытия тела.

Обычно функции-члены, определенные вне тела класса, не делают встроенными. Но объявить такую функцию встроенной можно, если явно добавить слово inline в объявление функции внутри тела класса или в ее определение вне тела, либо сделав то и другое одновременно. В следующем примере move() определена как встроенная функция-

```
inline void Screen::move( int r, int c )
{ // переместить курсор в абсолютную позицию
   if ( checkRange( r, c ) ) // позиция на экране задана корректно?
   {
      int row = (r-1) * _width; // смещение начала строки
      _cursor = row + c - 1;
   }
```

член класса Screen:

```
}
```

```
class Screen {
public:
   inline char get( int, int );
   // объявления других функций-членов не изменяются
```

Функция get(int, int) объявляется встроенной с помощью слова inline:

```
};
```

Определение функции следует после объявления класса. При этом слово inline можно

```
char Screen::get( int r, int c )
{
   move( r, c );   // устанавливаем _cursor
   return get();   // вызываем другую функцию-член get()
```

опустить:

```
}
```

Так как встроенные функции-члены должны быть определены в каждом исходном файле, где они вызываются, то встроенную функцию, не определенную в теле класса, следует поместить в тот же заголовочный файл, в котором определен ее класс. Например, представленные ранее определения move() и get() должны находиться в заголовочном файле Screen.h после определения класса Screen.

13.3.2. Доступ к членам класса

Говорят, что определение функции-члена принадлежит области видимости класса независимо от того, находится ли оно вне или внутри его тела. Отсюда следуют два вывода:

- в определении функции-члена могут быть обращения к любым членам класса, открытым или закрытым, и это не нарушает ограничений доступа;
- когда функция-член обращается к членам класса, операторы доступа “точка” и “стрелка” не необходимы.

```
#include <string>

void Screen::copy( const Screen &sobj )
{
   // если этот объект и объект sobj - одно и то же,
   // копирование излишне
   // мы анализируем указатель this (см. раздел 13.4)
   if ( this != &sobj )
   {
      _height = sobj._height;
      _width = sobj._width;
      _cursor = 0;

      // создаем новую строку;
      // ее содержимое такое же, как sobj._screen
      _screen = sobj._screen;
   }
}
```

Например:

```
#include "Screen.h"

int main()
{
   Screen s1;
   // Установить s1

   Screen s2;
   s2.copy(s1);

   // ...
}
```

Хотя _screen, _height, _width и _cursor являются закрытыми членами класса Screen, функция-член copy() работает с ними напрямую. Если при обращении к члену отсутствует оператор доступа, то считается, что речь идет о члене того класса, для которого функция-член вызвана. Если вызвать copy() следующим образом:

то параметр sobj внутри определения copy() соотносится с объектом s1 из функции main(). Функция-член copy() вызвана для объекта s2, стоящего перед оператором "точка". Для такого вызова члены _screen, _height, _width и _cursor, при обращении к которым внутри определения этой функции нет оператора доступа, – это члены объекта s2. В следующем разделе мы рассмотрим доступ к членам класса внутри определения функции-члена более подробно и, в частности, покажем, как для поддержки такого доступа применяется указатель this.

13.3.3. Закрытые и открытые функции-члены

Функцию-член можно объявить в любой из секций public, private или protected тела класса. Где именно это следует делать? Открытая функция-член задает операцию, которая может понадобиться пользователю. Множество открытых функций-членов

составляет *интерфейс* класса. Например, функции-члены `home()`, `move()` и `get()` класса `Screen` определяют операции, с помощью которых программа манипулирует объектами этого типа.

Поскольку мы прячем от пользователей внутреннее представление класса, объявляя его члены закрытыми, то для манипуляции объектами типа `Screen` необходимо предоставить открытые функции-члены. Такой прием – *сокрытие информации* – защищает написанный пользователем код от изменений во внутреннем представлении.

Внутреннее состояние объекта класса также защищено от случайных изменений. Все модификации объекта производятся с помощью небольшого набора функций, что существенно облегчает сопровождение и доказательство правильности программы.

До сих пор мы встречались лишь с функциями, поддерживающими доступ к закрытым членам только для чтения. Ниже приведены две функции `set()`, позволяющие

```
class Screen {
public:
   void set( const string &s );
   void set( char ch );
   // объявления других функций-членов не изменяются
```

пользователю модифицировать объект `Screen`. Добавим их объявления в тело класса:

```
};
```

```
void Screen::set( const string &s )
{ // писать в строку, начиная с текущей позиции курсора

   int space = remainingSpace();
   int len = s.size();
   if ( space < len ) {
      cerr << "Screen: warning: truncation: "
           << "space: " << space
           << "string length: " << len << endl;
      len = space;
   }

   _screen.replace( _cursor, len, s );
   _cursor += len - 1;
}

void Screen::set( char ch )
{
   if ( ch == '\0' )
      cerr << "Screen: warning: "
           << "null character (ignored).\n";
   else _screen[_cursor] = ch;
```

Далее следуют определения функций:

```
}
```

В реализации класса `Screen` мы предполагаем, что объект `Screen` не содержит двоичных нулей. По этой причине `set()` не позволяет записать на экран нуль.

Представленные до сих пор функции-члены были открытыми, их можно вызывать из любого места программы, а закрытые вызываются только из других функций-членов (или

друзей) класса, но не из программы, обеспечивая поддержку другим операциям в реализации абстракции класса. Примером может служить функция-член remainingSpace

```
class Screen {
public:
   // объявления других функций-членов не изменяются
private:
   inline int remainingSpace();
```

класса Screen(), использованная в set(const string&).

```
};
```

```
inline int Screen::remainingSpace()
{
   int sz = _width * _height;
   return ( sz - _cursor );
```

remainingSpace() сообщает, сколько места осталось на экране:

```
}
```

(Детально защищенные функции-члены будут рассмотрены в главе 17.)

Следующая программа предназначена для тестирования описанных к настоящему

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

int main() {
   Screen sobj(3,3); // конструктор определен в разделе 13.3.4
   string init("abcdefghi");
   cout << "Screen Object ( "
        << sobj.height() << ", "
        << sobj.width() << " )\n\n";

   // Задать содержимое экрана
   string::size_type initpos = 0;
   for ( int ix = 1; ix <= sobj.width(); ++ix )
      for ( int iy = 1; iy <= sobj.height(); ++iy )
      {
         sobj.move( ix, iy );
         sobj.set( init[ initpos++ ] );
      }

      // Напечатать содержимое экрана
      for ( int ix = 1; ix <= sobj.width(); ++ix )
      {
         for ( int iy = 1; iy <= sobj.height(); ++iy )
            cout << sobj.get( ix, iy );
         cout << "\n";
      }

      return 0;
```

моменту функций-членов:

```
}
```

Откомпилировав и запустив эту программу, мы получим следующее:

```
Screen Object ( 3, 3 )

abc
def
ghi
```

13.3.4. Специальные функции-члены

Существует специальная категория функций-членов, отвечающих за такие действия с объектами, как инициализация, присваивание, управление памятью, преобразование типов и уничтожение. Такие функции называются *конструкторами*. Они вызываются компилятором неявно каждый раз, когда объект класса определяется или создается оператором new. В объявлении конструктора его имя совпадает с именем класса. Вот, например, объявление конструктора класса Screen, в котором заданы значения по

```
class Screen {
public:
   Screen( int hi = 8, int wid = 40, char bkground = '#');
   // объявления других функций-членов не изменяются
};
```

умолчанию для параметров hi, wid и bkground:

```
Screen::Screen( int hi, int wid, char bk ) :
   _height( hi ),   // инициализировать _height значением hi
   _width( wid ),   // инициализировать _width значением wid
   _cursor ( 0 ),   // инициализировать _cursor нулем
   _screen( hi * wid, bk )  // размер экрана равен hi * wid
                            // все позиции инициализируются
                            // символом '#'
{ // вся работа проделана в списке инициализации членов
  // этот список обсуждается в разделе 14.5
}
```

Определение конструктора класса Screen выглядит так:

Каждый объявленный объект класса Screen автоматически инициализируется

```
Screen s1;                    // Screen(8,40,'#')
Screen *ps = new Screen( 20 ); // Screen(20,40,'#')

int main() {
   Screen s(24,80,'*');       // Screen(24,80,'*')
   // ...
}
```

конструктором:

(В главе 14 конструкторы, деструкторы и операторы присваивания рассматриваются более подробно. В главе 15 обсуждаются конвертеры и функции управления памятью.)

13.3.5. Функции-члены со спецификаторами const и volatile

Любая попытка модифицировать константный объект из программы обычно помечается

```
const char blank = ' ';
```

компилятором как ошибка. Например:

```
blank = '\n';    // ошибка
```

Однако объект класса, как правило, не модифицируется программой напрямую. Вместо этого вызывается та или иная открытая функция-член. Чтобы не было "покушений" на константность объекта, компилятор должен различать безопасные (те, которые не

```
const Screen blankScreen;
blankScreen.display();       // читает объект класса
```

изменяют объект) и небезопасные (те, которые пытаются это сделать) функции-члены:

```
blankScreen.set( '*' );      // ошибка: модифицирует объект класса
```

Проектировщик класса может указать, какие функции-члены не модифицируют объект,

```
class Screen {
public:
   char get() const { return _screen[_cursor]; }
   // ...
```

объявив их константными с помощью спецификатора `const`:

```
};
```

Для класса, объявленного как `const`, могут быть вызваны только те функции-члены, которые также объявлены со спецификатором `const`. Ключевое слово `const` помещается между списком параметров и телом функции-члена. Для константной функции-члена, определенной вне тела класса, это слово должно присутствовать как в объявлении, так и в определении:

```
class Screen {
public:
   bool isEqual( char ch ) const;
   // ...
private:
   string::size_type  _cursor;
   string             _screen;
   // ...
};

bool Screen::isEqual( char ch ) const
{
   return ch == _screen[_cursor];
}
```

Запрещено объявлять константную функцию-член, которая модифицирует члены класса.

```
class Screen {
public:
   int ok() const { return _cursor; }
   void error( int ival ) const { _cursor = ival; }
   // ...
private:
   string::size_type  _cursor;
   // ...
};
```

Например, в следующем упрощенном определении:

определение функции-члена `ok()` корректно, так как она не изменяет значения `_cursor`. В определении же `error()` значение `_cursor` изменяется, поэтому такая функция-член не может быть объявлена константной и компилятор выдает сообщение об ошибке:

```
error: cannot modify a data member within a const member function
ошибка: не могу модифицировать данные-члены внутри константной функции-члена
```

Если класс будет интенсивно использоваться, лучше объявить его функции-члены, не модифицирующие данных, константными. Однако наличие спецификатора `const` в объявлении функции-члена не предотвращает все возможные изменения. Такое объявление гарантирует лишь, что функции-члены не смогут изменять данные-члены, но если класс содержит указатели, то адресуемые ими объекты могут быть модифицированы константной функцией, не вызывая ошибки компиляции. Это часто приводит в недоумение начинающих программистов. Например:

```
#include <cstring>

class Text {
public:
   void bad( const string &parm ) const;
private:
   char *_text;
};

void Text::bad( const string &parm ) const
{
   _text = parm.c_str();   // ошибка: нельзя модифицировать _text

   for ( int ix = 0; ix < parm.size(); ++ix )
       _text[ix] = parm[ix];   // плохой стиль, но не ошибка

}
```

Модифицировать `_text` нельзя, но это объект типа `char*`, и символы, на которые он указывает, можно изменить внутри константной функции-члена класса `Text`. Функция-член `bad()` демонстрирует плохой стиль программирования. Константность функции-члена не гарантирует, что объекты внутри класса останутся неизменными после ее вызова, причем компилятор не поможет обнаружить такую ситуацию.

Константную функцию-член можно перегружать неконстантной функцией с тем же

```
class Screen {
public:
   char get(int x, int y);
   char get(int x, int y) const;
   // ...
```

списком параметров:

```
};
```

В этом случае наличие спецификатора `const` у объекта класса определяет, какая из двух

```
int main() {
   const Screen cs;
   Screen s;

   char ch = cs.get(0,0);   // вызывает константную функцию-член
   ch = s.get(0,0);         // вызывает неконстантную функцию-член
```

функций будет вызвана:

```
}
```

Хотя конструкторы и деструкторы не являются константными функциями-членами, они все же могут вызываться для константных объектов. Объект становится константным после того, как конструктор проинициализирует его, и перестает быть таковым, как только вызывается деструктор. Таким образом, объект со спецификатором `const` трактуется как константный с момента завершения работы конструктора и до вызова деструктора.

Функцию-член можно также объявить со спецификатором volatile (он был введен в разделе 3.13). Объект класса объявляется как volatile, если его значение изменяется способом, который не обнаруживается компилятором (например, если это структура данных, представляющая порт ввода/вывода). Для таких объектов вызываются только

```
class Screen {
public:
   char poll() volatile;
   // ...
};
```

функции-члены с тем же спецификатором, конструкторы и деструкторы:

```
char Screen::poll() volatile { ... }
```

13.3.6. Объявление mutable

При объявлении объекта класса Screen константным возникают некоторые проблемы. Предполагается, что после инициализации объекта Screen, его содержимое уже нельзя изменять. Но это не должно мешать нам читать содержимое экрана. Рассмотрим следующий константный объект класса Screen:

```
const Screen cs ( 5, 5 );
```

Если мы хотим прочитать символ, находящийся в позиции (3,4), то попробуем сделать

```
// прочитать содержимое экрана в позиции (3,4)
// Увы! Это не работает
cs.move( 3, 4 );
```

так:

```
char ch = cs.get();
```

Но такая конструкция не работает: move() – это не константная функция-член, и сделать

```
inline void Screen::move( int r, int c )
{
   if ( checkRange( r, c ) )
   {
      int row = (r-1) * _width;
      _cursor = row + c - 1;      // модифицирует _cursor
   }
```

ее таковой непросто. Определение move() выглядит следующим образом:

```
}
```

Обратите внимание, что move()изменяет член класса _cursor, следовательно, не может быть объявлена константной.

Но почему нельзя модифицировать _cursor для константного объекта класса Screen? Ведь _cursor – это просто индекс. Изменяя его, мы не модифицируем содержимое

экрана, а лишь пытаемся установить позицию внутри него. Модификация _cursor должна быть разрешена несмотря на то, что у класса Screen есть спецификатор const.

Чтобы разрешить модификацию члена класса, принадлежащего константному объекту, объявим его *изменчивым* (mutable). Член с таким спецификатором не бывает константным, даже если он член константного объекта. Его можно обновлять, в том числе функцией-членом со спецификатором const. Объявлению изменчивого члена

```
class Screen {
public:
   // функции-члены
private:
   string                      _screen;
   mutable string::size_type  _cursor; // изменчивый член
   short                       _height;
   short                       _width;
```

класса должно предшествовать ключевое слово mutable:

```
};
```

Теперь любая константная функция способна модифицировать _cursor, и move() может быть объявлена константной. Хотя move() изменяет данный член, компилятор не считает

```
// move() - константная функция-член
inline void Screen::move( int r, int c ) const
{
   // ...

   // правильно: константная функция-член может модифицировать члены
   // со спецификатором mutable
   _cursor = row + c - 1;
   // ...
```

это ошибкой.

```
}
```

Показанные в начале этого подраздела операции позиционирования внутри экрана теперь можно выполнить без сообщения об ошибке.

Отметим, что изменчивым объявлен только член _cursor, тогда как _screen, _height и _width не имеют спецификатора mutable, поскольку их значения в константном объекте класса Screen изменять нельзя.

Упражнение 13.3

```
Screen myScreen;
```

Объясните, как будет вести себя copy() при следующих вызовах:

```
myScreen.copy( myScreen );
```

Упражнение 13.4

К дополнительным перемещениям курсора можно отнести его передвижение вперед и назад на один символ. Из правого нижнего угла экрана курсор должен попасть в левый верхний угол. Реализуйте функции `forward()` и `backward()`.

Упражнение 13.5

Еще одной полезной возможностью является перемещение курсора вниз и вверх на одну строку. По достижении верхней или нижней строки экрана курсор не перепрыгивает на противоположный край; вместо этого подается звуковой сигнал, и курсор остается на месте. Реализуйте функции `up()` и `down()`. Для подачи сигнала следует вывести на стандартный вывод `cout` символ с кодом `'007'`.

Упражнение 13.6

Пересмотрите описанные функции-члены класса `Screen` и объявите те, которые сочтете нужными, константными. Объясните свое решение.

13.4. Неявный указатель this

```
int main() {
   Screen myScreen( 3, 3 ), bufScreen;

   myScreen.clear();
   myScreen.move( 2, 2 );
   myScreen.set( '*' );
   myScreen.display();

   bufScreen.resize( 5, 5 );
   bufScreen.display();
}
```

У каждого объекта класса есть собственная копия данных-членов. Например:

У объекта `myScreen` есть свои члены `_width`, `_height`, `_cursor` и `_screen`, а у объекта `bufScreen` – свои. Однако каждая функция-член класса существует в единственном экземпляре. Их и вызывают `myScreen` и `bufScreen`.

В предыдущем разделе мы видели, что функция-член может обращаться к членам своего класса, не используя операторы доступа. Так, определение функции `move()` выглядит

```
inline void Screen::move( int r, int c )
{
   if ( checkRange( r, c ) )      // позиция на экране задана корректно?
   {
      int row = (r-1) * _width;   // смещение строки
      _cursor = row + c - 1;
   }
}
```

следующим образом:

Если функция `move()` вызывается для объекта `myScreen`, то члены `_width` и `_height`, к которым внутри нее имеются обращения, – это члены объекта `myScreen`. Если же она вызывается для объекта `bufScreen`, то и обращения производятся к членам данного

объекта. Каким же образом _cursor, которым манипулирует move(), оказывается членом то myScreen, то bufScreen? Дело в указателе this.

Каждой функции-члену передается указатель на объект, для которого она вызвана, – this. В неконстантной функции-члене это указатель на тип класса, в константной – константный указатель на тот же тип, а в функции со спецификатором volatile указатель с тем же спецификатором. Например, внутри функции-члена move() класса Screen указатель this имеет тип Screen*, а в неконстантной функции-члене List – тип List*.

Поскольку this адресует объект, для которого вызвана функция-член, то при вызове move() для myScreen он указывает на объект myScreen, а при вызове для bufScreen – на объект bufScreen. Таким образом, член _cursor, с которым работает функция move(), в первом случае принадлежит объекту myScreen, а во втором – bufScreen.

Понять все это можно, если представить себе, как компилятор реализует объект this. Для его поддержки необходимо две трансформации:

```
// псевдокод, показывающий, как происходит расширение
// определения функции-члена
// ЭТО НЕ КОРРЕКТНЫЙ КОД C++
inline void Screen::move( Screen *this, int r, int c )
{
   if ( checkRange( r, c ) )
   {
      int row = (r-1) * this->_width;
      this->_cursor = row + c - 1;
   }
}
```

1. Изменить определение функции-члена класса, добавив дополнительный параметр:

   В этом определении использование указателя this для доступа к членам _width и _cursor сделано явным.

2. Изменение каждого вызова функции-члена класса с целью передачи одного дополнительного аргумента – адреса объекта, для которого она вызвана:

```
myScreen.move( 2, 2 );
```

   транслируется в

```
move( &myScreen, 2, 2 );
```

Программист может явно обращаться к указателю this внутри функции. Так, вполне

```
inline void Screen::home()
{
   this->_cursor = 0;
}
```

корректно, хотя и излишне, определить функцию-член home() следующим образом:

Однако бывают случаи, когда без такого обращения не обойтись, как мы видели на примере функции-члена copy() класса Screen. В следующем подразделе мы рассмотрим и другие примеры.

13.4.1. Когда использовать указатель this

Наша функция main() вызывает функции-члены класса Screen для объектов myScreen и bufScreen таким образом, что каждое действие – это отдельная инструкция. У нас есть возможность определить функции-члены так, чтобы конкатенировать их вызовы при обращении к одному и тому же объекту. Например, все вызовы внутри main() будут

```
int main() {
   // ...

   myScreen.clear().move( 2, 2 ), set( '*' ). display();
   bufScreen.reSize( 5, 5 ).display();
```

выглядеть так:

```
}
```

Именно так интуитивно представляется последовательность операций с экраном: очистить экран myScreen, переместить курсор в позицию (2,2), записать в эту позицию символ '*' и вывести результат.

Операторы доступа “точка” и “стрелка” левоассоциативны, т.е. их последовательность выполняется слева направо. Например, сначала вызывается myScreen.clear(), затем myScreen.move() и т.д. Чтобы myScreen.move() можно было вызвать после myScreen.clear(), функция clear() должна возвращать объект myScreen, для которого она была вызвана. Мы уже видели, что доступ к объекту внутри функции-члена

```
// объявление clear() находится в теле класса
// в нем задан аргумент по умолчанию bkground = '#'
Screen& Screen::clear( char bkground )
{ // установить курсор в левый верхний угол и очистить экран

   _cursor = 0;
   _screen.assign(       // записать в строку
      _screen.size(),    // size() символов
      bkground           // со значением bkground
   );

   // вернуть объект, для которого была вызвана функция
   return *this;
```

класса производится в помощью указателя this. Вот реализация clear():

```
}
```

Обратите внимание, что возвращаемый тип этой функции-члена – Screen& – ссылка на объект ее же класса. Чтобы конкатенировать вызовы, необходимо также пересмотреть реализацию move() и set(). Возвращаемый тип следует изменить с void на Screen&, а в определении возвращать *this.

Аналогично функцию-член display() можно написать так:

```
Screen& Screen::display()
{
   typedef string::size_type idx_type;

   for ( idx_type ix = 0; ix < _height; ++ix )
   { // для каждой строки

      idx_type offset = _width * ix; // смещение строки

      for ( idx_type iy = 0; iy < _width; ++iy )
          // для каждой колонки вывести элемент
          cout << _screen[ offset + iy ];

          cout << endl;
        }
        return *this;

}
```

```
// объявление reSize() находится в теле класса
// в нем задан аргумент по умолчанию bkground = '#'
Screen& Screen::reSize( int h, int w, char bkground )
{ // сделать высоту экрана равной h, а ширину - равной w
   // запомнить содержимое экрана
   string local(_screen);

   // заменить строку _screen
   _screen.assign(     // записать в строку
      h * w,           // h * w символов
      bkground         // со значением bkground
   );

   typedef string::size_type idx_type;
   idx_type local_pos = 0;

   // скопировать содержимое старого экрана в новый
   for ( idx_type ix = 0; ix < _height; ++ix )
   { // для каждой строки

       idx_type offset = w * ix; // смещение строки
       for ( idx_type iy = 0; iy < _width; ++iy )
          // для каждой колонки присвоить новое значение
          _screen[ offset + iy ] = local[ local_pos++ ];
   }

   _height = h;
   _width = w;
   // _cursor не меняется

   return *this;
```

А вот реализация `reSize()`:

```
}
```

Работа указателя `this` не исчерпывается возвратом объекта, к которому была применена функция-член. При рассмотрении `copy()` в разделе 13.3 мы видели и другой способ его использования:

```
void Screen::copy( const Screen& sobj )
{
   // если этот объект Screen и sobj - одно и то же,
   // копирование излишне
   if ( this != sobj )
   {
      // скопировать значение sobj в this
   }
}
```

Указатель this хранит адрес объекта, для которого была вызвана функция-член. Если адрес, на который ссылается sobj, совпадает со значением this, то sobj и this относятся к одному и тому же объекту, так что операция копирования не нужна. (Мы еще встретимся с этой конструкцией, когда будем рассматривать копирующий оператор присваивания в разделе 14.7.)

Упражнение 13.7

Указатель this можно использовать для модификации адресуемого объекта, а также для его замены другим объектом того же типа. Например, функция-член assign() класса

```
classType& classType::assign( const classType &source )
{
   if ( this != &source )
   {
      this->~classType();
      new (this) classType( source );
   }
   return *this;
}
```

classType выглядит так. Можете ли вы объяснить, что она делает?

Напомним, что ~classType – это имя деструктора. Оператор new выглядит несколько причудливо, но мы уже встречались с подобным в разделе 8.4.

Как вы относитесь к такому стилю программирования? Безопасна ли эта операция? Почему?

13.5. Статические члены класса

Иногда нужно, чтобы все объекты некоторого класса имели доступ к единственному глобальному объекту. Допустим, необходимо подсчитать, сколько их было создано; глобальным может быть указатель на процедуру обработки ошибок для класса или, скажем, указатель на свободную память для его объектов. В подобных случаях более эффективно иметь один глобальный объект, используемый всеми объектами класса, чем отдельные члены в каждом объекте. Хотя такой объект является глобальным, он существует лишь для поддержки реализации абстракции класса.

В этой ситуации приемлемым решением является статический член класса, который ведет себя как глобальный объект, принадлежащий своему классу. В отличие от других членов, которые присутствуют в каждом объекте как отдельные элементы данных, статический член существует в единственном экземпляре и связан с самим типом, а не с конкретным его объектом. Это разделяемая сущность, доступная всем объектам одного класса.

По сравнению с глобальным объектом у статического члена есть следующие преимущества:

- статический член не находится в глобальном пространстве имен программы, следовательно, уменьшается вероятность случайного конфликта имен с другими глобальными объектами;
- остается возможность сокрытия информации, так как статический член может быть закрытым, а глобальный объект – никогда.

Чтобы сделать член статическим, надо поместить в начале его объявления в теле класса ключевое слово `static`. К ним применимы все правила доступа к открытым, закрытым и защищенным членам. Например, для определенного ниже класса `Account` член

```
class Account {                  // расчетный счет
   Account( double amount, const string &owner );
   string owner() { return _owner; }
private:
   static double _interestRate;  // процентная ставка
   double        _amount;        // сумма на счету
   string        _owner;         // владелец
};
```

`_interestRate` объявлен как закрытый и статический типа `double`:

Почему `_interestRate` сделан статическим, а `_amount` и `_owner` нет? Потому что у всех счетов разные владельцы и суммы, но процентная ставка одинакова. Следовательно, объявление члена `_interestRate` статическим уменьшает объем памяти, необходимый для хранения объекта `Account`.

Хотя текущее значение `_interestRate` для всех счетов одинаково, но со временем оно может изменяться. Поэтому мы решили не объявлять этот член как `const`. Достаточно модифицировать его лишь один раз, и с этого момента все объекты `Account` будут видеть новое значение. Если бы у каждого объекта была собственная копия, то пришлось бы обновить их все, что неэффективно и является потенциальным источником ошибок.

В общем случае статический член инициализируется вне определения класса. Его имя во внешнем определении должно быть специфицировано именем класса. Вот так можно

```
// явная инициализация статического члена класса

#include "account.h"
```

инициализировать `_interestRate`:

```
double Account::_interestRate = 0.0589;
```

В программе может быть только одно определение статического члена. Это означает, что инициализацию таких членов следует помещать не в заголовочные файлы, а туда, где находятся определения невстроенных функций-членов класса.

В объявлении статического члена можно указать любой тип. Это могут быть константные объекты, массивы, объекты классов и т.д. Например:

```
#include <string>
class Account {
   // ...
private:
   static const string name;
};

const string Account::name( "Savings Account" );
```

Константный статический член целого типа инициализируется константой внутри тела класса: это особый случай. Если бы для хранения названия счета мы решили использовать массив символов вместо строки, то его размер можно было бы задать с

```
// заголовочный файл
class Account {
   //...
private:
   static const int nameSize = 16;
   static const string name[nameSize];
};

// исходный файл
const string Account::nameSize;   // необходимо определение члена
```

помощью константного члена типа int:

```
const string Account::name[nameSize] = "Savings Account";
```

Отметим, что константный статический член целого типа, инициализированный константой, – это *константное выражение*. Проектировщик может объявить такой статический член, если внутри тела класса возникает необходимость в именованной константе. Например, поскольку константный статический член nameSize является константным выражением, проектировщик использует его для задания размера члена-массива с именем name.

Даже если такой член инициализируется в теле класса, его все равно необходимо задать вне определения класса. Однако поскольку начальное значение уже задано в объявлении, то при определении оно не указывается.

Так как name – это массив (и не целого типа), его нельзя инициализировать в теле класса.

```
class Account {
   //...
private:
   static const int nameSize = 16;   // правильно: целый тип
   static const string name[nameSize] = "Savings Account";  // ошибка
```

Попытка поступить таким образом приведет к ошибке компиляции:

```
};
```

Член name должен быть инициализирован вне определения класса.

Обратите внимание, что член nameSize задает размер массива name в определении, находящемся вне тела класса:

```
const string Account::name[nameSize] = "Savings Account";
```

nameSize не квалифицирован именем класса Account. И хотя это закрытый член, определение name не приводит к ошибке. Как такое может быть? Определение статического члена аналогично определению функции-члена класса, которое может ссылаться на закрытые члены. Определение статического члена name находится в области видимости класса и может ссылаться на закрытые члены, после того как распознано квалифицированное имя Account::name. (Подробнее об области видимости класса мы поговорим в разделе 13.9.)

Статический член класса доступен функции-члену того же класса и без использования

```
inline double Account::dailyReturn()
{
   return( _interestRate / 365 * _amount );
```

соответствующих операторов:

```
}
```

Что же касается функций, не являющихся членами класса, то они могут обращаться к

```
class Account {
   // ...
private:
   friend int compareRevenue( Account&, Account* );
   // остальное без изменения
};

// мы используем ссылочный и указательный параметры,
// чтобы проиллюстрировать оба оператора доступа
int compareRevenue( Account &ac1, Account *ac2 );
{
   double ret1, ret2;
   ret1 = ac1._interestRate * ac1._amount;
   ret2 = ac2->_interestRate * ac2->_amount;
   // ...
```

статическому члену двумя способами. Во-первых, посредством операторов доступа:

```
}
```

Как ac1._interestRate, так и ac2->_interestRate относятся к статическому члену Account::_interestRate.

Поскольку есть лишь одна копия статического члена класса, до нее необязательно добираться через объект или указатель. Другой способ заключается в том, чтобы

```
// доступ к статическому члену с указанием квалифицированного имени
```

обратиться к статическому члену напрямую, квалифицировав его имя именем класса:

```
if ( Account::_interestRate < 0.05 )
```

Если обращение к статическому члену производится без помощи оператора доступа, то его имя следует квалифицировать именем класса, за которым следует оператор разрешения области видимости:

```
Account::
```

Это необходимо, поскольку такой член не является глобальным объектом, а значит, в глобальной области видимости отсутствует. Следующее определение дружественной

```
int compareRevenue( Account &ac1, Account *ac2 );
{
   double ret1, ret2;
   ret1 = Account::_interestRate * ac1._amount;
   ret2 = Account::_interestRate * ac2->_amount;
   // ...
```

функции compareRevenue эквивалентно приведенному выше:

```
}
```

Уникальная особенность статического члена – то, что он существует независимо от объектов класса, – позволяет использовать его такими способами, которые для нестатических членов недопустимы.

- статический член может принадлежать к типу того же класса, членом которого он является. Нестатические объявляются лишь как указатели или ссылки на объект

```
class Bar {
public:
   // ...
private:
   static Bar mem1;   // правильно
   Bar *mem2;         // правильно
   Bar mem3;          // ошибка
```

своего класса:

```
};
```

- статический член может выступать в роли аргумента по умолчанию для

```
extern int var;

class Foo {
private:
   int var;
   static int stcvar;
public:
   // ошибка: трактуется как Foo::var,
   // но ассоциированного объекта класса не существует
   int mem1( int = var );

   // правильно: трактуется как static Foo::stcvar,
   // ассоциированный объект и не нужен
   int mem2( int = stcvar );
   // правильно: трактуется как глобальная переменная var
   int mem3( int = :: var );
```

функции-члена класса, а для нестатического это запрещено:

```
};
```

13.5.1. Статические функции-члены

Функции-члены raiseInterest() и interest() обращаются к глобальному

```
class Account {
public:
   void raiseInterest( double incr );
   double interest() { return _interestRate; }
private:
   static double  _interestRate;
};

inline void Account::raiseInterest( double incr )
{
  _interestRate += incr;
}
```

статическому члену _interestRate:

Проблема в том, что любая функция-член должна вызываться с помощью оператора доступа к конкретному объекту класса. Поскольку приведенные выше функции обращаются только к статическому _interestRate, то совершенно безразлично, для какого объекта они вызываются. Нестатические члены при вызове этих функций не читаются и не модифицируются.

Поэтому лучше объявить такие функции-члены как статические. Это можно сделать

```
class Account {
public:
   static void raiseInterest( double incr );
   static double interest() { return _interestRate; }
private:
   static double  _interestRate;
};

inline void Account::raiseInterest( double incr )
{
  _interestRate += incr;
}
```

следующим образом:

Объявление статической функции-члена почти такое же, как и нестатической: в теле класса ему предшествует ключевое слово static, а спецификаторы const или volatile запрещены. В ее определении, находящемся вне тела класса, слова static быть не должно.

Такой функции-члену указатель this не передается, поэтому явное или неявное обращение к нему внутри ее тела вызывает ошибку компиляции. В частности, попытка обращения к нестатическому члену класса неявно требует наличия указателя this и, следовательно, запрещена. Например, представленную ранее функцию-член dailyReturn() нельзя объявить статической, поскольку она обращается к нестатическому члену _amount.

Статическую функцию-член можно вызвать для объекта класса, пользуясь одним из операторов доступа. Ее также можно вызвать непосредственно, квалифицировав ее имя, даже если никаких объектов класса не объявлено. Вот небольшая программа,

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

bool limitTest( double limit )
{
   // пока еще ни одного объекта класса Account не объявлено
   // правильно: вызов статической функции-члена
   return limit <= Account::interest() ;
}

int main() {
   double limit = 0.05;

   if ( limitTest( limit ) )
   {
      // указатель на статическую функцию-член
      // объявлен как обычный указатель
      void (*psf)(double) = &Account::raiseInterest;
      psf( 0.0025 );
   }

   Account ac1( 5000, "Asterix" );
   Account ac2( 10000, "Obelix" );
   if ( compareRevenue( ac1, &ac2 ) > 0 )
      cout << ac1.owner()
           << " is richer than "
           << ac2.owner() << "\n";
   else
      cout << ac1.owner()
           << " is poorer than "
           << ac2.owner() << "\n";
   return 0;
}
```

иллюстрирующая их применение:

Упражнение 13.8

Пусть дан класс Y с двумя статическими данными-членами и двумя статическими функциями-членами:

```
class X {
public:
   X( int i ) { _val = i; }
   int val() { return _val; }
private:
   int _val;
};

class Y {
public:
   Y( int i );
   static X xval();
   static int callsXval();
private:
   static X _xval;
   static int _callsXval;

};
```

Инициализируйте `_xval` значением 20, а `_callsXval` значением `0`.

Упражнение 13.9

Используя классы из упражнения 13.8, реализуйте обе статические функции-члена для класса `Y`. `callsXval()` должна подсчитывать, сколько раз вызывалась `xval()`.

Упражнение 13.10

```
// example.h
class Example {
public:
   static double rate = 6.5;

   static const int vecSize = 20;
   static vector<double> vec(vecSize);
};

// example.c
#include "example.h"
double Example::rate;
```

Какие из следующих объявлений и определений статических членов ошибочны? Почему?

```
vector<double> Example::vec;
```

13.6. Указатель на член класса

Предположим, что в нашем классе `Screen` определены четыре новых функции-члена: `forward()`, `back()`, `up()` и `down()`, которые перемещают курсор соответственно вправо, влево, вверх и вниз. Сначала мы должны объявить их в теле класса:

```
class Screen {
public:
   inline Screen& forward();
   inline Screen& back();
   inline Screen& end();
   inline Screen& up();
   inline Screen& down();
   // другие функции-члены не изменяются
private:
   inline int row();
   // другие функции-члены не изменяются
};
```

Функции-члены `forward()` и `back()` перемещают курсор на один символ. По достижении правого нижнего или левого верхнего угла экрана курсор переходит в противоположный угол.

```
inline Screen& Screen::forward()
{ // переместить _cursor вперед на одну экранную позицию

   ++_cursor;

   // если достигли конца экрана, перепрыгнуть в противоположный угол
   if ( _cursor == _screen.size() )
      home();

   return *this;
}

inline Screen& Screen::back()
{ // переместить _cursor назад на одну экранную позицию

   // если достигли начала экрана, перепрыгнуть в противоположный угол
   if ( _cursor == 0 )
      end();
   else
      --_cursor;

   return *this;
}
```

`end()` перемещает курсор в правый нижний угол экрана и является парной по отношению к функции-члену `home()`:

```
inline Screen& Screen::end()
{
   _cursor = _width * _height - 1;
   return *this;
}
```

Функции `up()` и `down()` перемещают курсор вверх и вниз на одну строку. По достижении верхней или нижней строки курсор остается на месте и подается звуковой сигнал:

```
const char BELL = '\007';

inline Screen& Screen::up()
{ // переместить _cursor на одну строку вверх
  // если уже наверху, остаться на месте и подать сигнал
  if ( row() == 1 ) // наверху?
     cout << BELL << endl;
  else
     _cursor -= _width;

  return *this;
}

inline Screen& Screen::down()
{
   if ( row() == _height ) //внизу?
      cout << BELL << endl;
   else
      _cursor += _width;

   return *this;
}
```

`row()` – это закрытая функция-член, которая используется в функциях `up()` и `down()`,

```
inline int Screen::row()
{ // вернуть текущую строку
   return ( _cursor + _width ) / height;
}
```

возвращая номер строки, где находится курсор:

Пользователи класса `Screen` попросили нас добавить функцию `repeat()`, которая

```
Screen &repeat( char op, int times )
{
   switch( op ) {
      case DOWN:    // n раз вызвать Screen::down()
         break;
      case DOWN:    // n раз вызвать Screen::up()
         break;
      // ...
   }
}
```

повторяет указанное действие `n` раз. Ее реализация могла бы выглядеть так:

Такая реализация имеет ряд недостатков. В частности, предполагается, что функции-члены класса `Screen` останутся неизменными, поэтому при добавлении или удалении функции-члена `repeat()` необходимо модифицировать. Вторая проблема – размер функции. Поскольку приходится проверять все возможные функции-члены, то исходный текст становится громоздким и неоправданно сложным.

В более общей реализации параметр `op` заменяется параметром типа указателя на функцию-член класса `Screen`. Теперь `repeat()` не должна сама устанавливать, какую

операцию следует выполнить, и всю инструкцию switch можно удалить. Определение и использование указателей на члены класса – тема последующих подразделов.

13.6.1. Тип члена класса

Указателю на функцию нельзя присвоить адрес функции-члена, даже если типы возвращаемых значений и списки параметров полностью совпадают. Например, переменная pfi – это указатель на функцию без параметров, которая возвращает значение типа int:

```
int (*pfi)();
```

```
int HeightIs();
```

Если имеются глобальные функции HeightIs() и WidthIs() вида:

```
int WidthIs();
```

```
pfi = HeightIs;
```

то допустимо присваивание pfi адреса любой из этих переменных:

```
pfi = WidthIs;
```

В классе Screen также определены две функции доступа, height() и width(), не

```
inline int Screen::height() { return _height; }
```

имеющие параметров и возвращающие значение типа int:

```
inline int Screen::width() { return _width; }
```

Однако попытка присвоить их переменной pfi является нарушением типизации и влечет

```
// неверное присваивание: нарушение типизации
pfi = &Screen::height;
```

ошибку компиляции:

В чем нарушение? У функций-членов есть дополнительный атрибут типа, отсутствующий у функций, не являющихся членами, – класс. Указатель на функцию-член должен соответствовать типу присваиваемой ему функции не в двух, а в трех отношениях: по типу и количеству формальных параметров; типу возвращаемого значения; типу класса, членом которого является функция.

Несоответствие типов между двумя указателями – на функцию-член и на обычную функцию – обусловлено их разницей в представлении. В указателе на обычную функцию хранится ее адрес, который можно использовать для непосредственного вызова. (Указатели на функции рассматривались в разделе 7.9.) Указатель же на функцию-член

должен быть сначала привязан к объекту или указателю на объект, чтобы получить this, и только после этого он применяется для вызова функции-члена. (В следующем подразделе мы покажем, как осуществить такую привязку.) Хотя для указателя на обычную функцию и для указателя на функцию-член используется один и тот же термин, их природа различна.

Синтаксис объявления указателя на функцию-член должен принимать во внимание тип класса. То же верно и в отношении указателей на данные-члены. Рассмотрим член _height класса Screen. Его полный тип таков: член класса Screen типа short. Следовательно, полный тип указателя на _height – это указатель на член класса Screen типа short:

```
short Screen::*
```

Определение указателя на член класса Screen типа short выглядит следующим образом:

```
short Screen::*ps_Screen;
```

Переменную ps_Screen можно инициализировать адресом _height:

```
short Screen::*ps_Screen = &Screen::_height;
```

или присвоить ей адрес _width:

```
short Screen::*ps_Screen = &Screen::_width;
```

Переменной ps_Screen разрешается присваивать указатель на _width или _height, так как они являются членами класса Screen типа short.

Несоответствие типов указателя на данные-члены и обычного указателя также связано с различием в их представлении. Обычный указатель содержит всю информацию, необходимую для обращения к объекту. Указатель на данные-члены следует сначала привязать к объекту или указателю на него, а лишь затем использовать для доступа к члену этого объекта. (В книге “Inside the C++ Object Model” ([LIPPMAN96a]) также описывается представление указателей на члены.)

Указатель на функцию-член определяется путем задания типа возвращаемого функцией значения, списка ее параметров и класса. Например, следующий указатель, с помощью которого можно вызвать функции height() и width(), имеет тип указателя на функцию-член класса Screen без параметров, которая возвращает значение типа int:

```
int (Screen::*)()
```

```
// всем указателям на функции-члены класса можно присвоить значение 0
int (Screen::*pmf1)() = 0;
int (Screen::*pmf2)() = &Screen::height;

pmf1 = pmf2;
```

Указатели на функции-члены можно объявлять, инициализировать и присваивать:

```
pmf2 = &Screen::width;
```

Использование typedef может облегчить чтение объявлений указателей на члены. Например, для типа “указатель на функцию-член класса Screen без параметров, которая возвращает ссылку на объект Screen”, т.е.

```
Screen& (Screen::*)()
```

```
typedef Screen& (Screen::*Action)();

Action default = &Screen::home;
```

Следующий typedef определяет Action как альтернативное имя:

```
Action next = &Screen::forward;
```

Тип “указатель на функцию-член” можно использовать для объявления формальных параметров и типа возвращаемого значения функции. Для параметра того же типа можно также указать значение аргумента по умолчанию:

```
Screen& action( Screen&, Action)();
```

action() объявлена как принимающая два параметра: ссылку на объект класса Screen и указатель на функцию-член Screen без параметров, которая возвращает ссылку на его

```
Screen meScreen;
typedef Screen& (Screen::*Action)();
Action default = &Screen::home;

extern Screen& action( Screen&, Sction = &Screen::display );

void ff()
{
   action( myScreen );
   action( myScreen, default );
   action( myScreen, &Screen::end );
}
```

объект. Вызвать action() можно любым из следующих способов:

В следующем подразделе обсуждается вызов функции-члена посредством указателя.

13.6.2. Работа с указателями на члены класса

К указателям на члены класса можно обращаться только с помощью конкретного объекта или указателя на объект типа класса. Для этого применяется любой из двух операторов доступа (.* для объектов класса и ссылок на них или ->* для указателей). Например, так вызывается функция-член через указатель на нее:

```
int (Screen::*pmfi)() = &Screen::height;
Screen& (Screen::*pmfS)( const Screen& ) = &Screen::copy;

Screen myScreen, *bufScreen;

// прямой вызов функции-члена
if ( myScreen.height() == bufScreen->height() )
   bufScreen->copy( myScreen );

// эквивалентный вызов по указателю
if ( (myScreen.*pmfi)() == (bufScreen->*pmfi)() )
   (bufScreen->*pmfS)( myScreen );
```

```
(myScreen.*pmfi)()
```

Вызовы

```
(bufScreen->*pmfi)();
```

требуют скобок, поскольку приоритет оператора вызова `()` выше, чем приоритет взятия указателя на функцию-член. Без скобок

```
myScreen.*pmfi()
```

интерпретируется как

```
myScreen.*(pmfi())
```

Это означает вызов функции `pmfi()` и привязку возвращенного ей значения к оператору `(.*)`. Разумеется, тип `pmfi` не поддерживает такого использования, так что компилятор выдаст сообщение об ошибке.

```
typedef short Screen::*ps_Screen;
Screen myScreen, *tmpScreen = new Screen( 10, 10 );

ps_Screen pH = &Screen::_height;
ps_Screen pW = &Screen::_width;

tmpScreen->*pH = myScreen.*pH;
```

Указатели на данные-члены используются аналогично:

```
tmpScreen->*pW = myScreen.*pW;
```

Приведем реализацию функции-члена `repeat()`, которую мы обсуждали в начале этого раздела. Теперь она будет принимать указатель на функцию-член:

```
typedef Screen& (Screen::Action)();

Screen& Screen::repeat( Action op, int times )
{
   for ( int i = 0; i < times; ++i )
      (this->*op)();
   return *this;
}
```

Параметр `op` – это указатель на функцию-член, которая должна вызываться `times` раз.

Если бы нужно было задать значения аргументов по умолчанию, то объявление `repeat()`

```
class Screen {
public:
   Screen &repeat( Action = &Screen::forward, int = 1 );
   // ...
};
```

выглядело бы следующим образом:

```
Screen myScreen;
myScreen.repeat();    // repeat( &Screen::forward, 1 );
```

А ее вызовы так:

```
myScreen.repeat( &Screen::down, 20 );
```

Определим таблицу указателей. В следующем примере `Menu` – это таблица указателей на функции-члены класса `Screen`, которые реализуют перемещение курсора. `CursorMovements` – перечисление, элементами которого являются номера в таблице

```
Action::Menu() = {
   &Screen::home,
   &Screen::forward,
   &Screen::back,
   &Screen::up,
   &Screen::down,
   &Screen::end
};

enum CursorMovements {
   HOME, FORWARD, BACK, UP, DOWN, END
};
```

`Menu`.

Можно определить перегруженную функцию-член `move()`, которая принимает параметр `CursorMovements` и использует таблицу `Menu` для вызова указанной функции-члена. Вот ее реализация:

```
Screen& Screen::move( CursorMovements cm )
{
   ( this->*Menu[ cm ] )();
   return *this;
}
```

У оператора взятия индекса ([]) приоритет выше, чем у оператора указателя на функцию-член (->*). Первая инструкция в `move()` сначала по индексу выбирает из таблицы `Menu` нужную функцию-член, которая и вызывается с помощью указателя `this` и оператора указателя на функцию-член. `move()` можно применять в интерактивной программе, где пользователь выбирает вид перемещения курсора из отображаемого на экране меню.

13.6.3. Указатели на статические члены класса

Между указателями на статические и нестатические члены класса есть разница. Синтаксис указателя на член класса не используется для обращения к статическому члену. Статические члены – это глобальные объекты и функции, принадлежащие классу. Указатели на них – это обычные указатели. (Напомним, что статической функции-члену не передается указатель `this`.)

Объявление указателя на статический член класса выглядит так же, как и для указателя на объект, не являющийся членом класса. Для разыменования указателя никакой объект

```
class Account {
public:
   static void raiseInterest( double incr );
   static double interest() { return _interestRate ; }
   double amount() { return _amount; }
private:
   static double  _interestRate;
   double         _amount;
   string         _owner;
};

inline void Account::raiseInterest( double incr )
{
 _interestRate += incr;
}
```

не требуется. Рассмотрим класс `Account`:

```
// это неправильный тип для &_interestRate
```

Тип `&_interestRate` – это `double*`:

```
double Account::*
```

Определение указателя на `&_interestRate` имеет вид:

```
// правильно: double*, а не double Account::*
double *pd = &Account::_interestRate;
```

Этот указатель разыменовывается так же, как и обычный, объект класса для этого не

```
Account unit;
// используется обычный оператор разыменования
```

требуется:

```
double daily = *pd / 365 * unit._amount;
```

Однако, поскольку _interestRate и _amount – закрытые члены, необходимо иметь статическую функцию-член interest() и нестатическую amount().

```
// правильно
```

Указатель на interest() – это обычный указатель на функцию:

```
double (*)()
```

```
// неправильно
```

а не на функцию-член класса Account:

```
double (Account::*)()
```

Определение указателя и косвенный вызов interest() реализуются так же, как и для

```
// правильно: double(*pf)(), а не double(Account::*pf)()
double(*pf)() = &Account::interest;
```

обычных указателей:

```
double daily = pf() / 365 * unit.amount();
```

Упражнение 13.11

К какому типу принадлежат члены _screen и _cursor класса Screen?

Упражнение 13.12

Определите указатель на член и инициализируйте его значением Screen::_screen; присвойте ему значение Screen::_cursor.

Упражнение 13.13

Определите typedef для каждой из функций-членов класса Screen.

Упражнение 13.14

Указатели на члены можно также объявлять как данные-члены класса. Модифицируйте определение класса `Screen` так, чтобы оно содержало указатель на его функцию-член того же типа, что `home()` и `end()`.

Упражнение 13.15

Модифицируйте имеющийся конструктор класса `Screen` (или напишите новый) так, чтобы он принимал параметр типа указателя на функцию-член класса `Screen`, для которой список формальных параметров и тип возвращаемого значения такие же, как у `home()` и `end()`. Реализуйте для этого параметра значение по умолчанию и используйте параметр для инициализации члена класса, описанного в упражнении 13.14. Напишите функцию-член `Screen`, позволяющую пользователю задать ее значение.

Упражнение 13.16

Определите перегруженный вариант `repeat()`, который принимает параметр типа `cursorMovements`.

13.7. Объединение – класс, экономящий память

Объединение – это специальный вид класса. Данные-члены хранятся в нем таким образом, что перекрывают друг друга. Все члены размещаются, начиная с одного и того же адреса. Для объединения отводится столько памяти, сколько необходимо для хранения самого большого его члена. В любой момент времени можно присвоить значение лишь одному такому члену.

Рассмотрим пример, иллюстрирующий использование объединения. Лексический анализатор, входящий в состав компилятора, разбивает программу на последовательность лексем. Так, инструкция

```
int i = 0;
```

преобразуется в последовательность из пяти лексем:

1. Ключевое слово `int`.
2. Идентификатор `i`.
3. Оператор `=`
4. Константа `0` типа `int`.
5. Точка с запятой.

Лексический анализатор передает эти лексемы синтаксическому анализатору, *парсеру*, который идентифицирует полученную последовательность. Полученная информация должна дать парсеру возможность распознать эту последовательность лексем как объявление. Для этого с каждой лексемой ассоциируется информация, позволяющая

```
Type ID Assign Constant Semicolon
```

парсеру увидеть следующее:

```
(Тип ИД Присваивание Константа Точка с запятой)
```

Далее парсер анализирует значения каждой лексемы. В данном случае он видит:

```
Type <==> int
ID <==> i

Constant <==> 0
```

Для Assign и Semicolon дополнительной информации не нужно, так как у них может быть только одно значение: соответственно := и ;.

Таким образом, в представлении лексемы могло бы быть два члена – token и value. token – это уникальный код, показывающий, что лексема имеет тип Type, ID, Assign, Constant или Semicolon, например 85 для ID и 72 для Semicolon.value содержит конкретное значение лексемы. Так, для лексемы ID в предыдущем объявлении value будет содержать строку "i", а для лексемы Type – некоторое представление типа int.

Представление члена value несколько проблематично. Хотя для любой отдельной лексемы в нем хранится всего одно значение, их типы для разных лексем могут различаться. Для лексемы ID в value хранится строка символов, а для Constant – целое число.

Конечно, для хранения данных нескольких типов можно использовать класс. Разработчик компилятора может объявить, что value принадлежит к типу класса, в котором для каждого типа данных есть отдельный член.

Применение класса решает проблему представления value. Однако для любой данной лексемы value имеет лишь один из множества возможных типов и, следовательно, будет задействован только один член класса, хотя памяти выделяется столько, сколько нужно для хранения всех членов. Чтобы память резервировалась только для нужного в данный

```
union TokenValue {
   char _cval;
   int _ival;
   char *_sval;
   double _dval;
```

момент члена, применяется объединение. Вот как оно определяется:

```
};
```

Если самым большим типом среди всех членов TokenValue является dval, то размер TokenValue будет равен размеру объекта типа double. По умолчанию члены объединения открыты. Имя объединения можно использовать в программе всюду, где

```
// объект типа TokenValue
TokenValue last_token;

// указатель на объект типа TokenValue
```

допустимо имя класса:

```
TokenValue *pt = new TokenValue;
```

Обращение к членам объединения, как и к членам класса, производится с помощью операторов доступа:

```
last_token._ival = 97;
char ch = pt->_cval;
```

```
union TokenValue {
public:
   char _cval;
   // ...
private:
   int priv;
}

int main() {
   TokenValue tp;
   tp._cval = '\n';   // правильно

   // ошибка: main() не может обращаться к закрытому члену
   //         TokenValue::priv
   tp.priv = 1024;
}
```

Члены объединения можно объявлять открытыми, закрытыми или защищенными:

У объединения не бывает статических членов или членов, являющихся ссылками. Его членом не может быть класс, имеющий конструктор, деструктор или копирующий

```
union illegal_members {
   Screen s;      // ошибка: есть конструктор
   Screen *ps;    // правильно
   static int is; // ошибка: статический член
   int &rfi;      // ошибка: член-ссылка
};
```

оператор присваивания. Например:

Для объединения разрешается определять функции-члены, включая конструкторы и деструкторы:

```
union TokenValue {
public:
   TokenValue(int ix) : _ival(ix) { }
   TokenValue(char ch) : _cval(ch) { }
   // ...
   int ival() { return _ival; }
   char cval() { return _cval; }
private:
   int _ival;
   char _cval;
   // ...
};

int main() {
   TokenValue tp(10);
   int ix = tp.ival();
   //...

}
```

```
enum TokenKind ( ID, Constant /* и другие типы лексем */ }
class Token {
public:
   TokenKind tok;
   TokenValue val;
};
```

Вот пример работы объединения TokenValue:

```
int lex() {
   Token curToken;
   char *curString;
   int curIval;

   // ...
   case ID:  // идентификатор
      curToken.tok = ID;
      curToken.val._sval = curString;
      break;

   case Constant:   // целая константа
      curToken.tok = Constant;
      curToken.val._ival = curIval;
      break;

   // ... и т.д.
}
```

Объект типа Token можно использовать так:

Опасность, связанная с применением объединения, заключается в том, что можно случайно извлечь хранящееся в нем значение, пользуясь не тем членом. Например, если в последний раз значение присваивалось _ival, то вряд ли понадобится значение, оказавшееся в _sval. Это, по всей вероятности, приведет к ошибке в программе.

Чтобы защититься от подобного рода ошибок, следует создать дополнительный объект, *дискриминант объединения*, определяющий тип значения, которое в данный момент

```
char *idVal;
// проверить значение дискриминанта перед тем, как обращаться к sval
if ( curToken.tok == ID )
```

хранится в объединении. В классе Token роль такого объекта играет член tok:

```
    idVal = curToken.val._sval;
```

При работе с объединением, являющимся членом класса, полезно иметь набор функций

```
#include <cassert>
// функции доступа к члену объединения sval
string Token::sval() {
   assert( tok==ID );
   return val._sval;
```

для каждого хранящегося в объединении типа данных:

```
}
```

Имя в определении объединения задавать необязательно. Если оно не используется в программе как имя типа для объявления других объектов, его можно опустить. Например, следующее определение объединения Token эквивалентно приведенному

```
class Token {
public:
   TokenKind tok;
   // имя типа объединения опущено
   union {
      char _cval;
      int _ival;
      char *_sval;
      double _dval;
   } val;
```

выше, но без указания имени:

```
};
```

Существует *анонимное объединение* – объединение без имени, за которым не следует определение объекта. Вот, например, определение класса Token, содержащее анонимное объединение:

```
class Token {
public:
   TokenKind tok;
   // анонимное объединение
   union {
      char _cval;
      int _ival;
      char *_sval;
      double _dval;
   };
};
```

К данным-членам анонимного объединения можно напрямую обращаться в той области видимости, в которой оно определено. Перепишем функцию `lex()`, используя

```
int lex() {
   Token curToken;
   char *curString;
   int curIval;

   // ... выяснить, что находится в лексеме
   // ... затем установить curToken
   case ID:
      curToken.tok = ID;
      curToken._sval = curString;
      break;
   case Constant:   // целая константа
      curToken.tok = Constant;
      curToken._ival = curIval;
      break;

   // ... и т.д.
```

предыдущее определение:

```
}
```

Анонимное объединение позволяет убрать один уровень доступа, поскольку обращение к его членам идет как к членам класса `Token`. У него не может быть закрытых или защищенных членов, а также функций-членов. Такое объединение, определенное в глобальной области видимости, должно быть объявлено в безымянном пространстве имен или иметь модификатор `static`.

13.8. Битовое поле – член, экономящий память

Для хранения заданного числа битов можно объявить член класса специального вида, называемый *битовым полем*. Он должен иметь целый тип данных, со знаком или без

```
class File {
   // ...
   unsigned int modified : 1;   // битовое поле
```

знака:

```
};
```

После идентификатора битового поля следует двоеточие, а за ним – константное выражение, задающее число битов. К примеру, `modified` – это поле из одного бита.

Битовые поля, определенные в теле класса подряд, по возможности упаковываются в соседние биты одного целого числа, делая хранение объекта более компактным. Так, в следующем объявлении пять битовых полей будут содержаться в одном числе типа

```
typedef unsigned int Bit;

class File {
public:
   Bit mode: 2;
   Bit modified: 1;
   Bit prot_owner: 3;
   Bit prot_group: 3;
   Bit prot_world: 3;
   // ...
```

`unsigned int`, ассоциированном с первым полем `mode`:

```
};
```

Доступ к битовому полю осуществляется так же, как к прочим членам класса. Скажем, к битовому полю, являющемуся закрытым членом класса, можно обратиться лишь из

```
void File::write()
{
   modified = 1;
   // ...
}

void File::close()
{
   if ( modified )
      // ... сохранить содержимое
```

функций-членов и друзей этого класса:

```
}
```

Вот простой пример использования битового поля длиной больше 1 (примененные здесь

```
enum { READ = 01, WRITE = 02 };   // режимы открытия файла

int main() {
   File myFile;

   myFile.mode |= READ;
   if ( myFile.mode & READ )
      cout << "myFile.mode is set to READ\n";
```

побитовые операции рассматривались в разделе 4.11):

```
}
```

Обычно для проверки значения битового поля-члена определяются встроенные функции-члены. Допустим, в классе `File` можно ввести члены `isRead()` и `isWrite()`:

```
inline int File::isRead() { return mode & READ; }
inline int File::isWrite() { return mode & WRITE; }
```

```
if ( myFile.isRead() ) /* ... */
```

С помощью таких функций-членов битовые поля можно сделать закрытыми членами класса `File`.

К битовому полю нельзя применять оператор взятия адреса (&), поэтому не может быть и указателя на подобные поля-члены. Кроме того, полю запрещено быть статическим членом.

В стандартной библиотеке C++ имеется шаблон класса `bitset`, который облегчает манипуляции с битовыми множествами. Мы рекомендуем использовать его вместо битовых полей. (Шаблон класса `bitset` и определенные в нем операции рассматривались в разделе 4.12.)

Упражнение 13.17

Перепишите примеры из этого подраздела так, чтобы в классе `File` вместо объявления и прямого манипулирования битовыми полями использовался класс `bitset` и его операторы.

13.9. Область видимости класса A

Тело класса определяет область видимости. Объявления членов класса внутри тела вводят их имена в область видимости класса.

Для обращения к ним применяются операторы доступа (точка и стрелка) и оператор разрешения области видимости (`::`). Когда употребляется оператор доступа, то предшествующее ему имя обозначает объект или указатель на объект типа класса, а следующее за ним имя должно находиться в области видимости этого класса. Аналогично при использовании оператора разрешения области видимости поиск имени, следующего за ним, идет в области видимости класса, имя которого стоит перед оператором. (В главах 17 и 18 мы увидим, что производный класс может обращаться к членам своих базовых.)

Однако применение операторов доступа или оператора разрешения области видимости нужно не всегда. Некоторые части программы сами по себе находятся в области видимости класса, и в них к членам класса можно обращаться напрямую. Одной из таких частей является само определение класса. Имя его члена можно использовать в теле после

```
class String {
public:
   typedef int index_type;

   // тип параметра - это на самом деле String::index_type
   char& operator[]( index_type )
```

объявления:

```
};
```

Порядок объявления членов класса в его теле важен: нельзя ссылаться на члены, которые будут объявлены позже. Например, если объявление оператора `operator[]()` находится

раньше объявления typedef index_type, то приведенное ниже объявление operator[]() оказывается ошибочным, поскольку в нем используется еще неизвестное

```
class String {
public:
   // ошибка: имя index_type не объявлено
   char &operator[]( index_type );

   typedef int index_type;
```

имя index_type:

```
};
```

Однако из этого правила есть два исключения. Первое касается имен, использованных в определениях встроенных функций-членов, второе – имен, применяемых как аргументы по умолчанию. Рассмотрим обе ситуации.

Разрешение имен в определениях встроенных функций-членов происходит в два этапа. Сначала объявление функции (т.е. тип возвращаемого значения и список параметров) обрабатывается в том месте, где оно встретилось в определении класса. Затем тело функции обрабатывается во всей области видимости, сразу после того, как были просмотрены объявления всех членов. Посмотрим на наш пример, в котором оператор

```
class String {
public:
   typedef int index_type;
   char &operator[]( index_type elem )
      { return _string[ elem ]; }
private:
   char *_string;
```

operator[]() определен как встроенный внутри тела класса:

```
};
```

На первом этапе просматриваются имена, использованные в объявлении operator[](), чтобы найти имя типа параметра index_type. Поскольку первый шаг выполняется тогда, когда в теле класса встретилось определение функции-члена, то имя index_type должно быть объявлено до определения operator[]().

Обратите внимание, что член _string объявлен в теле класса после определения operator[](). Это правильно, и _string не является в теле operator[]() необъявленным именем. Имена в телах функций-членов просматриваются на втором шаге разрешения имен в определениях встроенных функций-членов. Этот этап выполняется во всей области видимости класса, как если бы тела функций-членов обрабатывались последними, прямо перед закрытием тела класса, когда все его члены уже объявлены.

Аргументы по умолчанию также разрешаются на втором шаге. Например, в объявлении функции-члена clear() используется имя статического члена bkground, который определен позже:

```
class Screen {
public:
   // bkground относится к статическому члену,
   // объявленному позже в определении класса
   Screen& clear( char = bkground );
private:
   static const char bkground = '#';
};
```

Хотя такие аргументы в объявлениях функций-членов разрешаются во всей области видимости класса, программа будет считаться ошибочной, если он ссылается на нестатический член. Нестатический член должен быть привязан к объекту своего класса или к указателю на такой объект, иначе использовать его нельзя. Употребление подобных членов в качестве аргументов по умолчанию нарушает это ограничение. Если переписать

```
class Screen {
public:
   // ...
   // ошибка: bkground - нестатический член
   Screen& clear( char = bkground );
private:
   const char bkground = '#';
};
```

предыдущий пример так:

то имя аргумента по умолчанию разрешается нестатическим членом bkground, а это считается ошибкой.

Определения членов класса, появляющиеся вне его тела, – это еще один пример части программы, которая находится в области видимости класса. В ней имена членов распознаются несмотря на то, что оператор доступа или оператор разрешения области видимости при обращении к ним не применяется. Как же разрешаются имена в определениях членов?

Как правило, если такое определение появляется вне тела, то часть программы, следующая за именем определяемого члена, считается находящейся в области видимости класса вплоть до конца определения члена. Вынесем определение оператора

```
class String {
public:
   typedef int index_type;
   char& operator[]( index_type );
private:
   char *_string;
};

// в operator[]() есть обращения к index_type и _string
inline char& operator[]( index_type elem )
{
   return _string[ elem ];
}
```

operator[]() из класса String:

Обратите внимание, что в списке параметров встречается typedef index_type без квалифицирующего имени класса String::.Текст, следующий за именем члена String::operator[] и до конца определения функции, находится в области видимости класса. Объявленные в этой области типы рассматриваются при разрешении имен типов, использованных в списке параметров функции-члена.

Определения статических данных-членов также появляются вне определения класса. В них часть программы, следующая за именем статического члена вплоть до конца определения, считается находящейся в области видимости класса. Например, инициализатор статического члена может непосредственно, без соответствующих

```
class Account:
   // ...
private:
   static double _interestRate;
   static double initInterestRate();
};

// ссылается на Account::initInterest()
```

операторов, ссылаться на члены класса:

```
double Account::_interestRate = initInterest();
```

Инициализатор _interestRate вызывает статическую функцию-член Account::initInterest() несмотря на то, что ее имя не квалифицировано именем класса.

Не только инициализатор, но и все, что следует за именем статического члена _interestRate до завершающей точки с запятой, находится в области видимости класса Account. Поэтому в определении статического члена name может быть обращение к

```
class Account:
   // ...
private:
   static const int nameSize = 16;
   static const char name[nameSize];

// nameSize не квалифицировано именем класса Account
```

члену класса nameSize:

```
const char Account::name[nameSize] = "Savins Account";
```

Хотя член nameSize не квалифицирован именем класса Account, определение name не является ошибкой, так как оно находится в области видимости своего класса и может ссылаться на его члены после того, как компилятор прочитал Account::name.

В определении члена, которое появляется вне тела, часть программы перед определяемым именем не находится в области видимости класса. При обращении к члену в этой части следует пользоваться оператором разрешения области видимости. Например, если типом статического члена является typedef Money, определенный в классе Account, то имя Money должно быть квалифицировано, когда статический член данных определяется вне тела класса:

```
class Account {
   typedef double Money;
   //...
private:
   static Money _interestRate;
   static Money initInterest();
};
// Money должно быть квалифицировано именем класса Account::
Account::Money Account::_interestRate = initInterest();
```

С каждым классом ассоциируется отдельная область видимости, причем у разных классов эти области различны. К членам одного класса нельзя напрямую обращаться в определениях членов другого класса, если только один из них не является для второго базовым. (Наследование и базовые классы рассматриваются в главах 17 и 18.)

13.9.1. Разрешение имен в области видимости класса

Конечно, имена, используемые в области видимости класса, не обязаны быть именами членов класса. В процессе разрешения в этой области ведется поиск имен, объявленных и в других областях. Если имя, употребленное в области видимости класса, не разрешается именем члена класса, то компилятор ищет его в областях, включающих определение класса или члена. В этом подразделе мы покажем, как разрешаются имена, встречающиеся в области видимости класса.

Имя, использованное внутри определения класса (за исключением определений встроенных функций-членов и аргументов по умолчанию), разрешается следующим образом:

1. Просматриваются объявления членов класса, появляющиеся перед употреблением имени.
2. Если на шаге 1 разрешение не привело к успеху, то просматриваются объявления в пространстве имен перед определением класса. Напомним, что глобальная область видимости – это тоже область видимости пространства имен. (О пространствах имен речь шла в разделе 8.5.)

```
typedef double Money;
class Account {
   // ...
private:
   static Money _interestRate;
   static Money initInterest();
   // ...
```

Например:

```
};
```

Сначала компилятор ищет объявление `Money` в области видимости класса `Account`. При этом учитываются только те объявления, которые встречаются перед использованием `Money`. Поскольку таких объявлений нет, далее поиск ведется в глобальной области видимости. Объявление глобального `typedef Money` найдено, именно этот тип и используется в объявлениях `_interestRate` и `initInterest()`.

Имя, встретившееся в определении функции-члена класса, разрешается следующим образом:

1. Сначала просматриваются объявления в локальных областях видимости функции-члена. (О локальных областях видимости и локальных объявлениях говорилось в разделе 8.1.)
2. Если шаг 1 не привел к успеху, то просматриваются объявления для всех членов класса.
3. Если и этого оказалось недостаточно, просматриваются объявления в пространстве имен перед определением функции-члена.

```
int _height;

class Screen {
public:
   Screen( int _height ) {
      _height = 0;   // к чему относится _height? К параметру
   }
private:
   short _height;
};
```

Имена, встречающиеся в теле встроенной функции-члена, разрешаются так:

В поисках объявления имени _height, которое встретилось в определении конструктора Screen, компилятор просматривает локальную область видимости функции и находит его там. Следовательно, это имя относится к объявлению параметра.

Если бы такое объявление не было найдено, компилятор начал бы поиск в области видимости класса Screen, просматривая все объявления его членов, пока не встретится объявление члена _height. Говорят, что имя члена _height скрыто объявлением параметра конструктора, но его можно использовать в теле конструктора, если

```
int _height;

class Screen {
public:
   Screen( long _height ) {
      this->_height = 0;   // относится к Screen::_height
      // тоже правильно:
      // Screen::_height = 0;
   }
private:
   short _height;
};
```

квалифицировать имя члена именем его класса или явно использовать указатель this:

Если бы не были найдены ни объявление параметра, ни объявление члена, компилятор стал бы искать их в объемлющих областях видимости пространств имен. В нашем примере в глобальной области видимости просматриваются объявления, которые расположены перед определением класса Screen. В результате было бы найдено объявление глобального объекта _height. Говорят, что такой объект скрыт за

объявлением члена класса, однако его можно использовать в теле конструктора, если

```
int _height;

class Screen {
public:
   Screen( long _height ) {
      ::_height = 0;   // относится к глобальному объекту
   }
private:
   short _height;
};
```

квалифицировать оператором разрешения глобальной области видимости:

Если конструктор объявлен вне определения класса, то на третьем шаге разрешения имени просматриваются объявления в глобальной области видимости, которые встретились перед определением класса `Screen`, а также перед определением функции-

```
class Screen {
public:
   // ...
   void setHeight( int );
private:
   short _height;
};

int verify(int);

void Screen::setHeight( int var ) {
   // var: относится к параметру
   // _height: относится к члену класса
   // verify: относится к глобальной функции
   _height = verify( var );
}
```

члена:

Обратите внимание, что объявление глобальной функции `verify()` невидимо до определения класса `Screen`. Однако на третьем шаге разрешения имени просматриваются объявления в областях видимости пространств имен, видимые перед определением члена, поэтому нужное объявление обнаруживается.

Имя, встретившееся в определении статического члена класса, разрешается следующим образом:

1. Просматриваются объявления всех членов класса.
2. Если шаг 1 не привел к успеху, то просматриваются объявления, расположенные в областях видимости пространств имен перед определением статического члена, а не только предшествующие определению класса.

Упражнение 13.18

Назовите те части программы, которые находятся в области видимости класса.

Упражнение 13.19

Назовите те части программы, которые находятся в области видимости класса и для которых при разрешении имен просматривается полная область (т.е. принимаются во внимание все члены, объявленные в теле класса).

Упражнение 13.20

К каким объявлениям относится имя Type при использовании в теле класса Exersise и в определении его функции-члена setVal()? (Напоминаем, что разные вхождения могут относиться к разным объявлениям.) К каким объявлениям относится имя initVal при

```
typedef int Type;
Type initVal();

class Exercise {
public:
   // ...
   typedef double Type;
   Type setVal( Type );
   Type initVal();
private:
   int val;
};

Type Exercise::setVal( Type parm ) {
   val = parm + initVal();
}
```

употреблении в определении функции-члена setVal()?

Определение функции-члена setVal() ошибочно. Можете ли вы сказать, почему? Внесите необходимые изменения, чтобы в классе Exercise использовался глобальный typedef Type и глобальная функция initVal().

13.10. Вложенные классы A

Класс, объявленный внутри другого класса, называется *вложенным*. Он является членом объемлющего класса, а его определение может находиться в любой из секций public, private или protected объемлющего класса.

Имя вложенного класса известно в области видимости объемлющего класса, но ни в каких других областях. Это означает, что оно не конфликтует с таким же именем, объявленным в объемлющей области видимости. Например:

```
class Node { /* ... */ }

class Tree {
public:
   // Node инкапсулирован внутри области видимости класса Tree
   // В этой области Tree::Node скрывает ::Node
   class Node {...};

   // правильно: разрешается в пользу вложенного класса: Tree::Node
   Node *tree;
};

// Tree::Node невидима в глобальной области видимости
// Node разрешается в пользу глобального объявления Node
Node *pnode;

class List {
public:
   // Node инкапсулирован внутри области видимости класса List
   // В этой области List::Node скрывает ::Node
   class Node {...};

   // правильно: разрешается в пользу вложенного класса: List::Node
   Node *list;
};
```

```
// Не идеально, будем улучшать
class List {
public:
   class ListItem {
      friend class List;       // объявление друга
      ListItem( int val=0 );   // конструктор
      ListItem *next;          // указатель на собственный класс
      int value;
   };
   // ...
private:
   ListItem *list;
   ListItem *at_end;
```

Для вложенного класса допустимы такие же виды членов, как и для невложенного:

```
};
```

Закрытым называется член, который доступен только в определениях членов и друзей класса. У объемлющего класса нет права доступа к закрытым членам вложенного. Чтобы в определениях членов `List` можно было обращаться к закрытым членам `ListItem`, класс `ListItem` объявляет `List` как друга. Равно и вложенный класс не имеет никаких специальных прав доступа к закрытым членам объемлющего класса. Если бы нужно было разрешить `ListItem` доступ к закрытым членам класса `List`, то в объемлющем классе `List` следовало бы объявить вложенный класс как друга. В приведенном выше примере этого не сделано, поэтому `ListItem` не может обращаться к закрытым членам `List`.

Объявление `ListItem` открытым членом класса `List` означает, что вложенный класс можно использовать как тип во всей программе, в том числе и за пределами определений членов и друзей класса. Например:

```
// правильно: объявление в глобальной области видимости
List::ListItem *headptr;
```

Это дает более широкую область видимости, чем мы планировали. Вложенный `ListItem` поддерживает абстракцию класса `List` и не должен быть доступен во всей программе.

```
// Не идеально, будем улучшать
class List {
public:
   // ...
private:
   class ListItem {
      // ...
   };
   ListItem *list;
   ListItem *at_end;
```

Поэтому лучше объявить вложенный класс `ListItem` закрытым членом `List`:

```
};
```

Теперь тип `ListItem` доступен только из определений членов и друзей класса `List`, поэтому все члены класса `ListItem` можно сделать открытыми. При таком подходе объявление `List` как друга `ListItem` становится ненужным. Вот новое определение

```
// так лучше
class List {
public:
   // ...
private:
   // Теперь ListItem закрытый вложенный тип
   class ListItem {
   // а его члены открыты
   public:
      ListItem( int val=0 );
      ListItem *next;
      int value;
   };
   ListItem *list;
   ListItem *at_end;
```

класса `List`:

```
};
```

Конструктор `ListItem` не задан как встроенный внутри определения класса и, следовательно, должен быть определен вне него. Но где именно? Конструктор класса `ListItem` не является членом `List` и, значит, не может быть определен в теле последнего; его нужно определить в глобальной области видимости – той, которая содержит определение объемлющего класса. Когда функция-член вложенного класса не определяется как встроенная в теле, она должна быть определена вне самого внешнего из объемлющих классов.

Вот как могло бы выглядеть определение конструктора `ListItem`. Однако показанный ниже синтаксис в глобальной области видимости некорректен:

```
class List {
public:
   // ...
private:
   class ListItem {
   public:
      ListItem( int val=0 );
      // ...
   };
};

// ошибка: ListItem вне области видимости
ListItem:: ListItem( int val ) { ... }
```

Проблема в том, что имя `ListItem` отсутствует в глобальной области видимости. При использовании его таким образом следует указывать, что `ListItem` – вложенный класс в области видимости `List`. Это делается путем квалификации имени `ListItem` именем

```
// имя вложенного класса квалифировано именем объемлющего
List::ListItem::ListItem( int val ) {
   value = val;
   next = 0;
}
```

объемлющего класса. Следующая конструкция синтаксически правильна:

Заметим, что квалифицировано только имя вложенного класса. Первый квалификатор `List::` именует объемлющий класс и квалифицирует следующее за ним имя вложенного `ListItem`. Второе вхождение `ListItem` – это имя конструктора, а не вложенного класса.

```
// ошибка: конструктор называется ListItem, а не List::ListItem
List::ListItem::List::ListItem( int val ) {
   value = val;
   next = 0;
}
```

В данном определении имя члена некорректно:

Если бы внутри `ListItem` был объявлен статический член, то его определение также следовало бы поместить в глобальную область видимости. Имя этого члена могло бы выглядеть так:

```
int List::ListItem::static_mem = 1024;
```

Обратите внимание, что функции-члены и статические данные-члены не обязаны быть открытыми членами вложенного класса для того, чтобы их можно было определить вне его тела. Закрытые члены `ListItem` также определяются в глобальной области видимости.

Вложенный класс разрешается определять вне тела объемлющего. Например, определение `ListItem` могло бы находиться и в глобальной области видимости:

```
class List {
public:
   // ...
private:
   // объявление необходимо
   class ListItem;
   ListItem *list;
   ListItem *at_end;
};

// имя вложенного класса квалифицировано именем объемлющего класса
class List::ListItem {
public:
   ListItem( int val=0 );
   ListItem *next;
   int value;
};
```

В глобальном определении имя вложенного `ListItem` должно быть квалифицировано именем объемлющего класса `List`. Заметьте, что объявление `ListItem` в теле `List` опустить нельзя. Определение вложенного класса не может быть задано в глобальной области видимости, если предварительно оно не было объявлено членом объемлющего класса. Но при этом вложенный класс не обязательно должен быть открытым членом объемлющего.

Пока компилятор не увидел определения вложенного класса, разрешается объявлять лишь указатели и ссылки на него. Объявления членов `list` и `at_end` класса `List` правильны несмотря на то, что `ListItem` определен в глобальной области видимости, поскольку оба члена – указатели. Если бы один из них был объектом, то его объявление в классе `List`

```
class List {
public:
   // ...
private:
   // объявление необходимо
   class ListItem;
   ListItem *list;
   ListItem at_end;   // ошибка: неопределенный вложенный класс ListItem
};
```

привело бы к ошибке компиляции:

Зачем определять вложенный класс вне тела объемлющего? Возможно, он поддерживает некоторые детали реализации `ListItem`, а нам нужно скрыть их от пользователей класса `List`. Поэтому мы помещаем определение вложенного класса в заголовочный файл, содержащий интерфейс `List`. Таким образом, определение `ListItem` может находиться лишь внутри исходного файла, включающего реализацию класса `List` и его членов.

Вложенный класс можно сначала объявить, а затем определить в теле объемлющего. Это позволяет иметь во вложенных классах члены, ссылающиеся друг на друга:

```
class List {
public:
   // ...
private:
   // объявление List::ListItem
   class ListItem;
   class Ref {
      // pli имеет тип List::ListItem*
      ListItem *pli;
   };
   определение List::ListItem
   class ListItem {
      // pref имеет тип List::Ref*
      Ref *pref;
   };
};
```

Если бы `ListItem` не был объявлен перед определением класса `Ref`, то объявление члена `pli` было бы ошибкой.

Вложенный класс не может напрямую обращаться к нестатическим членам объемлющего, даже если они открыты. Любое такое обращение должно производиться через указатель,

```
class List {
public:
   int init( int );
private:
   class List::ListItem {
   public:
      ListItem( int val=0 );
      void mf( const List & );
      int value;
   };
};

List::ListItem::ListItem { int val )
{
   // List::init() - нестатический член класса List
   // должен использоваться через объект или указатель на тип List
   value = init( val );   // ошибка: неверное использование init
```

ссылку или объект объемлющего класса. Например:

```
};
```

При использовании нестатических членов класса компилятор должен иметь возможность идентифицировать объект, которому принадлежит такой член. Внутри функции-члена класса `ListItem` указатель `this` неявно применяется лишь к его членам. Благодаря неявному `this` мы знаем, что член `value` относится к объекту, для которого вызван конструктор. Внутри конструктора `ListItem` указатель `this` имеет тип `ListItem*`. Для доступа же к функции-члену `init()` нужен объект типа `List` или указатель типа `List*`.

Следующая функция-член `mf()` обращается к `init()` с помощью параметра-ссылки. Таким образом, `init()` вызывается для объекта, переданного в аргументе функции:

```
void List::ListItem::mf( List &i1 ) {
   memb = i1.init();   // правильно: обращается к init() по ссылке
}
```

Хотя для доступа к нестатическим членам объемлющего класса нужен объект, указатель или ссылка, к статическим его членам, именам типов и элементам перечисления вложенный класс может обращаться напрямую (если, конечно, эти члены открыты). Имя

```
class List {
public:
   typedef int (*pFunc)();
   enum ListStatus { Good, Empty, Corrupted };
   //...
private:
   class ListItem {
   public:
      void check_status();
      ListStatus status;     // правильно
      pFunc action;   // правильно
      // ...
   };
   // ...
};
```

типа – это либо имя `typedef`, либо имя перечисления, либо имя класса. Например:

`pFunc`, `ListStatus` и `ListItem` – все это вложенные имена типов в области видимости объемлющего класса `List`. К ним, а также к элементам перечисления `ListStatus` можно

```
void List::ListItem::check_status()
{
   ListStatus s = status;
   switch ( s ) {
      case Empty: ...
      case Corrupted: ...
      case Good: ...
   }
}
```

обращаться в области видимости класса `ListItem` даже без квалификации:

Вне области видимости `ListItem` и `List` при обращении к статическим членам, именам типов и элементам перечисления объемлющего класса требуется оператор разрешения

```
List::pFunc myAction;  // правильно
```

области видимости:

```
List::ListStatus stat = List::Empty;   // правильно
```

При обращении к элементам перечисления мы не пишем:

```
List::ListStatus::Empty
```

поскольку они доступны непосредственно в той области видимости, в которой определено само перечисление. Почему? Потому что с ним, в отличие от класса, не связана отдельная область.

13.10.1. Разрешение имен в области видимости вложенного класса

Посмотрим, как разрешаются имена в определениях вложенного класса и его членов.

Имя, встречающееся в определении вложенного класса (кроме тех, которые употребляются во встроенных функциях-членах и аргументах по умолчанию) разрешается следующим образом:

1. Просматриваются члены вложенного класса, расположенные перед употреблением имени.
2. Если шаг 1 не привел к успеху, то просматриваются объявления членов объемлющего класса, расположенные перед употреблением имени.
3. Если и этого недостаточно, то просматриваются объявления, расположенные в области видимости пространства имен перед определением вложенного класса.

```
enum ListStatus { Good, Empty, Corrupted };
class List {
public:
   // ...
private:
   class ListItem {
   public:
      // Смотрим в:
      // 1) List::ListItem
      // 2) List
      // 3) глобальной области видимости
      ListStatus status;   // относится к глобальному перечислению
      // ...
   };
   // ...
```

Например:

```
};
```

Сначала компилятор ищет объявление `ListStatus` в области видимости класса `ListItem`. Поскольку его там нет, поиск продолжается в области видимости `List`, а затем в глобальной. При этом во всех трех областях просматриваются только объявления, предшествующие использованию `ListStatus`. В конце концов находится глобальное объявление перечисления `ListStatus` – оно и будет типом, использованным в объявлении `status`.

Если вложенный класс `ListItem` определен в глобальной области видимости, вне тела объемлющего класса `List`, то все члены `List` уже были объявлены:

```
class List {
private:
   class ListItem {
   //...
public:
   enum ListStatus { Good, Empty, Corrupted };
   // ...
};

class List::ListItem {
public:
   // Смотрим в:
   // 1) List::ListItem
   // 2) List
   // 3) глобальной области видимости
   ListStatus status;   // относится к глобальному перечислению
   // ...

};
```

При разрешении имени `ListStatus` сначала просматривается область видимости класса `ListItem`. Поскольку там его нет, поиск продолжается в области видимости `List`. Так как полное определение класса `List` уже встречалось, просматриваются все члены этого класса. Вложенное перечисление `ListStatus` найдено несмотря даже на то, что оно объявлено после объявления `ListItem`. Таким образом, `status` объявляется как указатель на данное перечисление в классе `List`. Если бы в `List` не было члена с таким именем, поиск был бы продолжен в глобальной области видимости среди тех объявлений, которые предшествуют определению класса `ListItem`.

Имя, встретившееся в определении функции-члена вложенного класса, разрешается следующим образом:

1. Сначала просматриваются локальные области видимости функции-члена.
2. Если шаг 1 не привел к успеху, то просматриваются объявления всех членов вложенного класса.
3. Если имя еще не найдено, то просматриваются объявления всех членов объемлющего класса.
4. Если и этого недостаточно, то просматриваются объявления, появляющиеся в области видимости пространства имен перед определением функции-члена.

Какое объявление относится к имени `list` в определении функции-члена `check_status()` в следующем фрагменте кода:

```
class List {
public:
   enum ListStatus { Good, Empty, Corrupted };
   // ...
private:
   class ListItem {
   public:
      void check_status();
      ListStatus status;   // правильно
      //...
   };
   ListItem *list;
};

int list = 0;
void List::ListItem::check_status()
{
   int value = list;  // какой list?
}
```

Весьма вероятно, что при использовании `list` внутри `check_status()` программист имел в виду глобальный объект:

- и `value`, и глобальный объект `list` имеют тип `int`. Член `List::list` объявлен как указатель и не может быть присвоен `value` без явного приведения типа;
- `ListItem` не имеет прав доступа к закрытым членам объемлющего класса, в частности `list`;
- `list` – это нестатический член, и обращение к нему в функциях-членах `ListItem` должно производиться через объект, указатель или ссылку.

Однако, несмотря на все это, имя `list`, встречающееся в функции-члене `check_status()`, разрешается в пользу члена `list` класса `List`. Напоминаем, что если имя не найдено в области видимости вложенного `ListItem`, то далее просматривается область видимости объемлющего класса, а не глобальная. Член `list` в `List` скрывает глобальный объект. А так как использование указателя `list` в `check_status()` недопустимо, то выводится сообщение об ошибке.

Права доступа и совместимость типов проверяются только после того, как имя разрешено. Если при этом обнаруживается ошибка, то выдается сообщение о ней и дальнейший поиск объявления, которое было бы лучше согласовано с именем, уже не производится. Для доступа к глобальному объекту `list` следует использовать оператор

```
void List::ListItem::check_status()
{
   int value = ::list;  // правильно
}
```

разрешения области видимости:

Если бы функция-член `check_status()` была определена как встроенная в теле класса `ListItem`, то последнее объявление привело бы к выдаче сообщения об ошибке из-за того, что имя `list` не объявлено в глобальной области видимости:

```
class List {
public:
   // ...
private:
   class ListItem {
   public:
      // ошибка: нет видимого объявления для ::list
      void check_status() { int value = ::lis; }
      //...
   };
   ListItem *list;
   // ...
};
```

```
int list = 0;
```

Глобальный объект `list` объявлен после определения класса `List`. Во встроенной функции-члене, определенной внутри тела класса, рассматриваются только те глобальные объявления, которые были видны перед определением объемлющего класса. Если же определение `check_status()` следует за определением `List`, то рассматриваются глобальные объявления, расположенные перед ним, поэтому будет найдено глобальное определение объекта `list`.

Упражнение 13.21

В главе 11 был приведен пример программы, использующей класс `iStack`. Измените его, объявив классы исключений `pushOnFull` и `popOnEmpty` открытыми вложенными в `iStack`. Модифицируйте соответствующим образом определение класса `iStack` и его функций-членов, а также определение `main()`.

13.11. Классы как члены пространства имен A

Представленные до сих пор классы определены в области видимости глобального пространства имен. Но их можно определять и в объявленных пользователем пространствах. Имя класса, определенного таким образом, доступно только в области видимости этого пространства, т.е. оно не конфликтует с именами, объявленными в

```
namespace cplusplus_primer {
   class Node { /* ... */ };
}
namespace DisneyFeatureAnimation {
   class Node { /* ... */ };
}
Node *pnode;    // ошибка: Node не видно в глобальной области видимости

// правильно: объявляет nodeObj как объект
// квалифицированного типа DisneyFeatureAnimation::Node
DisneyFeatureAnimation::Node nodeObj;

// using-объявление делает Node видимым в глобальной области видимости
using cplusplus_primer::Node;
```

других пространствах имен. Например:

```
Node another;    // cplusplus_primer::Node
```

Как было показано в двух предыдущих разделах, член класса (функция-член, статический член или вложенный класс) может быть определен вне его тела. Если мы реализуем библиотеку и помещаем определения наших классов в объявленное пользователем пространство имен, то где расположить определения членов, находящиеся вне тел своих классов? Их можно разместить либо в пространстве имен, которое содержит определение самого внешнего класса, либо в одном из объемлющих его пространств. Это дает

```
// --- primer.h ---
namespace cplusplus_primer {
   class List {
   // ...
   private:
      class ListItem {
      public:
         void check_status();
         int action();
         // ...
      };
   };
}
// --- primer.C ---
#include "primer.h"

namespace cplusplus_primer {
   // правильно: check_status() определено в том же пространстве имен,
   // что и List
   void List::ListItem::check_status() { }
}

// правильно: action() определена в глобальной области видимости
//            в пространстве имен, объемлющем определение класса List
// Имя члена квалифицировано именем пространства
```

возможность организовать код библиотеки следующим образом:

```
int cplusplus_primer::List::ListItem::action() { }
```

Члены вложенного класса `ListItem` можно определить в пространстве имен `cplusplus_primer`, которое содержит определение `List`, или в глобальном пространстве, включающем определение `cplusplus_primer`. В любом случае имя члена в определении должно быть квалифицировано именами объемлющих классов и объявленных пользователем пространств, вне которых находится объявление члена.

Как происходит разрешение имени в определении члена, которое находится в

```
int cplusplus_primer::List::ListItem::action() {
   int local = someVal;
   // ...
```

объявленном пользователем пространстве? Например, как будет разрешено `someVal`:

```
}
```

Сначала просматриваются локальные области видимости в определении функции-члена, затем поиск продолжается в области видимости `ListItem`, затем – в области видимости `List`. До этого момента все происходит так же, как в процессе разрешения имен, описанном в разделе 13.10. Далее просматриваются объявления из пространства `cplusplus_primer` и наконец объявления в глобальной области видимости, причем во

внимание принимаются только те, которые расположены до определения функции-члена

```
// --- primer.h ---
namespace cplusplus_primer {
   class List {
      // ...
   private:
      class ListItem {
      public:
         int action();
         // ...
      };
   };
   const int someVal = 365;
}

// --- primer.C ---
#include "primer.h"

namespace cplusplus_primer {

   int List::ListItem::action() {
      // правильно: cplusplus_primer::someVal
      int local = someVal;

      // ошибка: calc() еще не объявлена
      double result = calc( local );
      // ...
   }

   double calc(int) { }
   // ...
```

action():

```
}
```

Определение пространства имен `cplusplus_primer` не является непрерывным. Определения класса `List` и объекта `someVal` размещены в первом его разделе, который находится в заголовочном файле `primer.h`. Определение функции `calc()` появляется в определении пространства имен, расположенном в файле реализации `primer.C`. Использование `calc()` внутри `action()` ошибочно, так как она объявлена после использования. Если `calc()` – часть интерфейса `cplusplus_primer`, ее следовало бы

```
// --- primer.h ---
namespace cplusplus_primer {
   class List {
      // ...
   }
   const int someVal = 365;
   double calc(int);
```

объявить в той части данного пространства, которая находится в заголовочном файле:

```
}
```

Если же `calc()` используется только в `action()` и не является частью интерфейса пространства имен, то ее нужно объявить перед `action()`, чтобы можно было ссылаться на нее внутри определения `action()`.

Здесь прослеживается аналогия с процессом поиска объявлений в глобальной области видимости, о котором мы говорили в предыдущих разделах: объявления, предшествующие определению члена, принимаются во внимание, тогда как следующие за ним игнорируются.

Довольно просто запомнить, в каком порядке просматриваются области видимости при поиске имени из определения функции, расположенного вне определения класса. Имена, которыми квалифицировано имя члена, указывают порядок рассмотрения пространств. Например, имя `action()` в предыдущем примере квалифицируется так:

```
cplusplus_primer::List::ListItem::action()
```

Квалификаторы `cplusplus_primer::List::ListItem::` записаны в порядке, обратном тому, в котором просматриваются имена областей видимости классов и пространств имен. Сначала поиск ведется в области `ListItem`, затем продолжается в объемлющем классе `List` и наконец в пространстве `cplusplus_primer`, предшествующем той области, в которой находится определение `action()`. Во время поиска в любой области видимости класса просматриваются все объявления членов, а в любом пространстве имен – только те объявления, которые встречались перед определением члена.

Класс, определенный в области видимости пространства имен, потенциально виден во всей программе. Если заголовочный файл `primer.h` включен в несколько исходных файлов, то имя `cplusplus_primer::List` везде относится к одному и тому же классу. Класс – это сущность, для которой в программе может быть более одного определения. Определение класса должно присутствовать один раз в каждом исходном файле, где определяются или используются сам класс или его члены. Однако оно должно быть одинаковым во всех файлах, где встречается, поэтому его следует помещать в заголовочный файл, например `primer.h`. Затем такой файл можно включать в любой исходный, где определяются или используются члены класса. Это предотвратит несоответствия в случае, когда определение класса записывается более одного раза.

Невстроенные функции-члены и статические данные-члены класса в пространстве имен – это также программные сущности. Однако они могут быть определены лишь один раз во всей программе. Поэтому их определения помещаются не в заголовочный, а в отдельный исходный файл типа `primer.C`.

Упражнение 13.22

Используя класс `iStack`, определенный в упражнении 13.21, объявите классы

```
namespace LibException {
   class pushOnFull{ };
   class popOnEmpty{ };
```

исключений `pushOnFull` и `popOnEmpty` как члены пространства имен `LibException`:

```
}
```

а сам `iStack` – членом пространства имен `Container`. Модифицируйте соответствующим образом определение данного класса и его функций-членов, а также определение `main()`.

13.12. Локальные классы A

Класс, определенный внутри тела функции, называется *локальным*. Он виден только в той локальной области, где определен. Не существует синтаксиса, позволяющего обратиться к члену такого класса, в отличие от вложенного, извне локальной области видимости, содержащей его определение. Поэтому функции-члены локального класса должны определяться внутри определения самого класса. На практике это ограничивает их сложность несколькими строками кода; помимо всего прочего, такой код становится трудно читать.

Поскольку невозможно определить член локального класса в области видимости пространства имен, то в таком классе не бывает статических членов.

Класс, вложенный в локальный, может быть определен вне определения объемлющего класса, но только в локальной области видимости, содержащей это определение. Имя вложенного класса в таком определении должно быть квалифицировано именем

```
void foo( int val )
{
   class Bar {
   public:
      int barVal;
      class nested;   // объявление вложенного класса обязательно
   };

   // определение вложенного класса
   class Bar::nexted {
      // ...
   };
```

объемлющего класса. Объявление вложенного класса в объемлющем нельзя опускать:

```
}
```

У объемлющей функции нет никаких специальных прав доступа к закрытым членам локального класса. Разумеется, это можно обойти, объявив ее другом данного класса. Однако необходимость делать его члены закрытыми вообще сомнительна, поскольку часть программы, из которой разрешается обратиться к нему, весьма ограничена. Локальный класс инкапсулирован в своей локальной области видимости. Дальнейшая инкапсуляция путем сокрытия информации не требуется: вряд ли на практике найдется причина, по которой не все члены локального класса должны быть открыты.

У локального класса, как и у вложенного, ограничен доступ к именам из объемлющей области видимости. Он может обратиться только к именам типов, статических переменных и элементов перечислений, определенных в объемлющих локальных областях. Например:

```
int a, val;
void foo( int val )
{
   static int si;
   enum Loc { a = 1024, b };
   class Bar {
   public:
      Loc locVal;   // правильно
      int barVal;
      void fooBar ( Loc l = a ) {  // правильно: Loc::a
         barVal = val;       // ошибка: локальный объект
         barVal = ::val;     // правильно: глобальный объект
         barVal = si;        // правильно: статический локальный объект
         locVal = b;         // правильно: элемент перечисления
      }
   };
   // ...
}
```

Имена в теле локального класса разрешаются лексически путем поиска в объемлющих областях видимости объявлений, предшествующих определению такого класса. При разрешении имен, встречающихся в телах его функций-членов, сначала просматривается область видимости класса, а только потом – объемлющие области,

Как всегда, если первое найденное объявление таково, что употребление имени оказывается некорректным, поиск других объявлений не производится. Несмотря на то что использование `val` в `fooBar()` выше является ошибкой, глобальная переменная `val` не будет найдена, если только ее имени не предшествует оператор разрешения глобальной области видимости.

14. Инициализация, присваивание и уничтожение класса

В этой главе мы детально изучим автоматическую инициализацию, присваивание и уничтожение объектов классов в программе. Для поддержки инициализации служит конструктор – определенная проектировщиком функция (возможно, перегруженная), которая автоматически применяется к каждому объекту класса перед его первым использованием. Парная по отношению к конструктору функция, деструктор, автоматически применяется к каждому объекту класса по окончании его использования и предназначена для освобождения ресурсов, захваченных либо в конструкторе класса, либо на протяжении его жизни.

По умолчанию как инициализация, так и присваивание одного объекта класса другому выполняются почленно, т.е. путем последовательного копирования всех членов. Хотя этого обычно достаточно, при некоторых обстоятельствах такая семантика оказывается неадекватной. Тогда проектировщик класса должен предоставить специальный копирующий конструктор и копирующий оператор присваивания. Самое сложное в поддержке этих функций-членов – понять, что они должны быть написаны.

14.1. Инициализация класса

```
class Data {
public:
   int ival;
   char *ptr;
};
```

Рассмотрим следующее определение класса:

Чтобы безопасно пользоваться объектом класса, необходимо правильно инициализировать его члены. Однако смысл этого действия для разных классов различен. Например, может ли `ival` содержать отрицательное значение или нуль? Каковы правильные начальные значения обоих членов класса? Мы не ответим на эти вопросы, не понимая абстракции, представляемой классом. Если с его помощью описываются служащие компании, то `ptr`, вероятно, указывает на фамилию служащего, а `ival` – его уникальный номер. Тогда отрицательное или нулевое значения ошибочны. Если же класс представляет текущую температуру в городе, то допустимы любые значения `ival`. Возможно также, что класс `Data` представляет *строку со счетчиком ссылок*: в таком случае `ival` содержит текущее число ссылок на строку по адресу `ptr`. При такой абстракции `ival` инициализируется значением 1; как только значение становится равным `0`, объект класса уничтожается.

Мнемонические имена класса и обоих его членов сделали бы, конечно, его назначение более понятным для читателя программы, но не дали бы никакой дополнительной

информации компилятору. Чтобы компилятор понимал наши намерения, мы должны предоставить одну или несколько перегруженных функций инициализации – *конструкторов*. Подходящий конструктор выбирается в зависимости от множества начальных значений, указанных при определении объекта. Например, любая из приведенных ниже инструкций представляет корректную инициализацию объекта класса

```
Data dat01( "Venus and the Graces", 107925 );
Data dat02( "about" );
Data dat03( 107925 );
```

`Data`:

```
Data dat04;
```

Бывают ситуации (как в случае с `dat04`), когда нам нужен объект класса, но его начальные значения мы еще не знаем. Возможно, они станут известны позже. Однако начальное значение задать необходимо, хотя бы такое, которое показывает, что разумное начальное значение еще не присвоено. Другими словами, инициализация объекта иногда сводится к тому, чтобы показать, что он еще *не* инициализирован. Большинство классов предоставляют специальный *конструктор по умолчанию*, для которого не требуется задавать начальных значений. Как правило, он инициализирует объект таким образом, чтобы позже можно было понять, что реальной инициализации еще не проводилось.

Обязан ли наш класс `Data` иметь конструктор? Нет, поскольку все его члены открыты. Унаследованный из языка C механизм поддерживает явную инициализацию,

```
int main()
{
   // local1.ival = 0; local1.ptr = 0
   Data local1 = { 0, 0 };

   // local2.ival = 1024;
   // local3.ptr = "Anna Livia Plurabelle"
   Data.local2 - { 1024, "Anna Livia Plurabelle" };

   // ...
}
```

аналогичную используемой при инициализации массивов:

Значения присваиваются позиционно, на основе порядка, в котором объявляются данные-члены. Следующий пример приводит к ошибке компиляции, так как `ival` объявлен перед

```
// ошибка: ival = "Anna Livia Plurabelle";
//         ptr = 1024
```

`ptr`:

```
Data.local2 - { "Anna Livia Plurabelle", 1024 };
```

Явная инициализация имеет два основных недостатка. Во-первых, она может быть применена лишь для объектов классов, все члены которых открыты (т.е. эта инициализация не поддерживает инкапсуляции данных и абстрактных типов – их не было в языке C, откуда она заимствована). А во-вторых, такая форма требует

вмешательства программиста, что увеличивает вероятность появления ошибок (забыл включить список инициализации или перепутал порядок следования инициализаторов в нем).

Так нужно ли применять явную инициализацию вместо конструкторов? Да. Для некоторых приложений более эффективно использовать список для инициализации больших структур постоянными значениями. К примеру, мы можем таким образом построить палитру цветов или включить в текст программы фиксированные координаты вершин и значения в узлах сложной геометрической модели. В подобных случаях инициализация выполняется во время загрузки, что сокращает затраты времени на запуск конструктора, даже если он определен как встроенный. Это особенно удобно при работе с глобальными объектами[1].

Примечание [O.A.4]: Нумерация сносок сбита.

Однако в общем случае предпочтительным методом инициализации является конструктор, который гарантированно будет вызван компилятором для каждого объекта до его первого использования. В следующем разделе мы познакомимся с конструкторами детально.

14.2. Конструктор класса

Среди других функций-членов конструктор выделяется тем, что его имя совпадает с

```
class Account {
public:
   // конструктор по умолчанию ...
   Account();
   // ...
private:
   char *_name;
   unsigned int _acct_nmbr;
   double _balance;
};
```

именем класса. Для объявления конструктора по умолчанию мы пишем[2]:

Единственное синтаксическое ограничение, налагаемое на конструктор, состоит в том, что он не должен иметь тип возвращаемого значения, даже void. Поэтому следующие

```
// ошибки: у конструктора не может быть типа возвращаемого значения
void     Account::Account() { ... }
```

объявления ошибочны:

```
Account* Account::Account( const char *pc ) { ... }
```

---

1 Более подробное обсуждение этой темы с примерами и приблизительными оценками производительности см. в [LIPPMAN96a].

2 В реальной программе мы объявили бы член _name как имеющий тип string. Здесь он объявлен как C-строка, чтобы отложить рассмотрение вопроса об инициализации членов класса до раздела 14.4.

Количество конструкторов у одного класса может быть любым, лишь бы все они имели разные списки формальных параметров.

Откуда мы знаем, сколько и каких конструкторов определить? Как минимум, необходимо присвоить начальное значение каждому члену, который в этом нуждается. Например, номер счета либо задается явно, либо генерируется автоматически таким образом, чтобы гарантировать его уникальность. Предположим, что он будет создаваться автоматически. Тогда мы должны разрешить инициализировать оставшиеся два члена _name и _balance:

```
Account( const char *name, double open_balance );
```

Объект класса Account, инициализируемый конструктором, можно объявить следующим образом:

```
Account newAcct( "Mikey Matz", 0 );
```

Если же есть много счетов, для которых начальный баланс равен 0, то полезно иметь конструктор, задающий только имя владельца и автоматически инициализирующий _balance нулем. Один из способов сделать это – предоставить конструктор вида:

```
Account( const char *name );
```

Другой способ – включить в конструктор с двумя параметрами значение по умолчанию, равное нулю:

```
Account( const char *name, double open_balance = 0.0 );
```

Оба конструктора обладают необходимой пользователю функциональностью, поэтому оба решения приемлемы. Мы предпочитаем использовать аргумент по умолчанию, поскольку в такой ситуации общее число конструкторов класса сокращается.

Нужно ли поддерживать также задание одного лишь начального баланса без указания имени клиента? В данном случае спецификация класса явно запрещает это. Наш конструктор с двумя параметрами, из которых второй имеет значение по умолчанию, предоставляет полный интерфейс для указания начальных значений тех членов класса

```
class Account {
public:
   // конструктор по умолчанию ...
   Account();

   // имена параметров в объявлении указывать необязательно
   Account( const char*, double=0.0 );

   const char* name() { return name; }
   // ...
private:
   // ...
};
```

Account, которые могут быть инициализированы пользователем:

Ниже приведены два примера правильного определения объекта класса Account, где конструктору передается один или два аргумента:

```
int main()
{
   // правильно: в обоих случаях вызывается конструктор
   // с двумя параметрами
   Account acct( "Ethan Stern" );
   Account *pact = new Account( "Michael Lieberman", 5000 );

   if ( strcmp( acct.name(), pact->name() ))
      // ...

}
```

C++ требует, чтобы конструктор применялся к определенному объекту до его первого использования. Это означает, что как для acct, так и для объекта, на который указывает pact, конструктор будет вызван перед проверкой в инструкции if.

Компилятор перестраивает нашу программу, вставляя вызовы конструкторов. Вот как, по

```
// псевдокод на C++,
// иллюстрирующий внутреннюю вставку конструктора
int main()
{
   Account acct;
   acct.Account::Account("Ethan Stern", 0.0);

   // ...

}
```

всей вероятности, будет модифицировано определение acct внутри main():

Конечно, если конструктор определен как встроенный, то он подставляется в точке вызова.

Обработка оператора new несколько сложнее. Конструктор вызывается только тогда, когда он успешно выделил память. Модификация определения pact в несколько

```
// псевдокод на C++,
// иллюстрирующий внутреннюю вставку конструктора при обработке new
int main()
{
   // ...

   Account *pact;
   try {
        pact = _new( sizeof( Account ));
        pact->Acct.Account::Account(
                 "Michael Liebarman", 5000.0);
   }
   catch( std::bad_alloc ) {
          // оператор new закончился неудачей:
          // конструктор не вызывается
   }
   // ...

}
```

упрощенном виде выглядит так:

```
// в общем случае эти формы эквивалентны
Account acct1( "Anna Press" );
Account acct2 = Account( "Anna Press" );
```

Существует три в общем случае эквивалентных формы задания аргументов конструктора:

```
Account acct3 = "Anna Press";
```

Форма acct3 может использоваться только при задании единственного аргумента. Если аргументов два или более, мы рекомендуем пользоваться формой acct1, хотя допустима

```
// рекомендуемая форма вызова конструктора
```

и acct2.

```
Account acct1( "Anna Press" );
```

Новички часто допускают ошибку при объявлении объекта, инициализированного

```
// увы! работает не так, как ожидалось
```

конструктором по умолчанию:

```
Account newAccount();
```

Эта инструкция компилируется без ошибок. Однако при попытке использовать объект в

```
// ошибка компиляции ...
```

таком контексте:

```
if ( ! newAccount.name() ) ...
```

компилятор не сможет применить к функции нотацию доступа к членам класса.

```
// определяет функцию newAccount,
// а не объект класса
```

Определение

```
Account newAccount();
```

интерпретируется компилятором как определение функции без параметров, которая возвращает объект типа Account. Правильное объявление объекта класса,

```
// правильно: определяется объект класса ...
```

инициализируемого конструктором по умолчанию, не содержит пустых скобок:

```
Account newAccount;
```

Определять объект класса, не указывая списка фактических аргументов, можно в том случае, если в нем либо объявлен конструктор по умолчанию, либо вообще нет

объявлений конструкторов. Если в классе объявлен хотя бы один конструктор, то не разрешается определять объект класса, не вызывая ни одного из них. В частности, если в классе определен конструктор, принимающий один или более параметров, но не определен конструктор по умолчанию, то в каждом определении объекта такого класса должны присутствовать необходимые аргументы. Можно возразить, что не имеет смысла определять конструктор по умолчанию для класса Account, поскольку не бывает счетов без имени владельца. В пересмотренной версии класса Account такой конструктор

```
class Account {
public:
   // имена параметров в объявлении указывать необязательно
   Account( const char*, double=0.0 );

   const char* name() { return name; }
   // ...
private:
   // ...
```

исключен:

```
};
```

Теперь при объявлении каждого объекта Account в конструкторе *обязательно* надо указать как минимум аргумент типа С-строки, но это скорее всего бессмысленно. Почему? Контейнерные классы (например, vector) требуют, чтобы для класса помещаемых в них элементов был либо задан конструктор по умолчанию, либо вообще никаких конструкторов. Аналогичная ситуация имеет место при выделении динамического массива объектов класса. Так, следующая инструкция вызвала бы ошибку

```
// ошибка: требуется конструктор по умолчанию для класса Account
```

компиляции для новой версии Account:

```
Account *pact = new Account[ new_client_cnt ];
```

На практике часто требуется задавать конструктор по умолчанию, если имеются какие-либо другие конструкторы.

А если для класса нет разумных значений по умолчанию? Например, класс Account требует задавать для любого объекта фамилию владельца счета. В таком случае лучше всего установить состояние объекта так, чтобы было видно, что он еще не

```
// конструктор по умолчанию для класса Account
inline Account::
Account() {
   _name = 0;
   _balance = 0.0;
   _acct_nmbr = 0;
```

инициализирован корректными значениями:

```
}
```

Однако в функции-члены класса Account придется включить проверку целостности объекта перед его использованием.

Существует и альтернативный синтаксис: *список инициализации членов*, в котором через запятую указываются имена и начальные значения. Например, конструктор по

```
// конструктор по умолчанию класса Account с использованием
// списка инициализации членов
inline Account::
Account()
       : _name(0),
         _balance( 0.0 ), _acct_nmbr( 0 )
```

умолчанию можно переписать следующим образом:

```
{}
```

Такой список допустим только в определении, но не в объявлении конструктора. Он помещается между списком параметров и телом конструктора и отделяется двоеточием. Вот как выглядит наш конструктор с двумя параметрами при частичном использовании

```
inline Account::
Account( const char* name, double opening_bal )
       : _balance( opening_bal )
{
      _name = new char[ strlen(name)+1 ];
      strcpy( _name, name );

      _acct_nmbr = get_unique_acct_nmbr();
```

списка инициализации членов:

```
}
```

`get_unique_acct_nmbr()` – это не являющаяся открытой функция-член, которая возвращает гарантированно не использованный ранее номер счета.

Конструктор нельзя объявлять с ключевыми словами `const` или `volatile` (см. раздел

```
class Account {
public:
   Account() const;     // ошибка
   Account() volatile;  // ошибка
   // ...
```

13.3.5), поэтому приведенные записи неверны:

```
};
```

Это не означает, что объекты класса с такими спецификаторами запрещено инициализировать конструктором. Просто к объекту применяется подходящий конструктор, причем без учета спецификаторов в объявлении объекта. Константность объекта класса устанавливается после того, как работа по его инициализации завершена, и пропадает в момент вызова деструктора. Таким образом, объект класса со спецификатором `const` считается константным с момента завершения работы конструктора до момента запуска деструктора. То же самое относится и к спецификатору `volatile`.

Рассмотрим следующий фрагмент программы:

```
// в каком-то заголовочном файле
extern void print( const Account &acct );

// ...

int main()
{
   // преобразует строку "oops" в объект класса Account
   // с помощью конструктора Account::Account( "oops", 0.0 )
   print( "oops" );

   // ...
}
```

По умолчанию конструктор с одним параметром (или с несколькими – при условии, что все параметры, кроме первого, имеют значения по умолчанию) играет роль оператора преобразования. В этом фрагменте программы конструктор Account неявно применяется компилятором для трансформации литеральной строки в объект класса Account при вызове print(), хотя в данной ситуации такое преобразование не нужно.

Непреднамеренные неявные преобразования классов, например трансформация "oops" в объект класса Account, оказались источником трудно обнаруживаемых ошибок. Поэтому в стандарт C++ было добавлено ключевое слово explicit, говорящее

```
class Account {
public:
   explicit Account( const char*, double=0.0 );
};
```

компилятору, что такие преобразования не нужны:

Данный модификатор применим только к конструктору. (Операторы преобразования и слово explicit обсуждаются в разделе 15.9.2.)

14.2.1. Конструктор по умолчанию

Конструктором по умолчанию называется конструктор, который можно вызывать, не задавая аргументов. Это не значит, что такой конструктор не может принимать аргументов; просто с каждым его формальным параметром ассоциировано значение по

```
// все это конструкторы по умолчанию
Account::Account() { ... }
iStack::iStack( int size = 0 ) { ... }
```

умолчанию:

```
Complex::Complex(double re=0.0, double im=0.0) { ... }
```

Когда мы пишем:

```
int main()
{
   Account acct;
   // ...
}
```

то компилятор сначала проверяет, определен ли для класса Account конструктор по умолчанию. Возникает одна из следующих ситуаций:

1. Такой конструктор определен. Тогда он применяется к acct.
2. Конструктор определен, но не является открытым. В данном случае определение acct помечается компилятором как ошибка: у функции main() нет прав доступа.
3. Конструктор по умолчанию не определен, но есть один или несколько конструкторов, требующих задания аргументов. Определение acct помечается как ошибка: слишком мало аргументов у конструктора.
4. Нет ни конструктора по умолчанию, ни какого-либо другого. Определение считается корректным, acct не инициализируется, конструктор не вызывается.

Пункты 1 и 3 должны быть уже достаточно понятны (если это не так, перечитайте данную главу) Посмотрим более внимательно на пункты 2 и 4.

Допустим, что все члены класса Account объявлены открытыми и не объявлено никакого

```
class Account {
public:
   char          *_name;
   unsigned int   _acct_nmbr;
   double         _balance;
```

конструктора:

```
};
```

В таком случае при определении объекта класса Account специальной инициализации не производится. Начальные значения всех трех членов зависят только от контекста, в котором встретилось определение. Например, для статических объектов гарантируется, что все их члены будут обнулены (как и для объектов, не являющихся экземплярами

```
// статический класс хранения
// вся ассоциированная с объектом память обнуляется

Account global_scope_acct;
static Account file_scope_acct;

Account foo()
{
   static Account local_static_acct;
   // ...
```

классов):

```
}
```

Однако объекты, определенные локально или распределенные динамически, в начальный

```
// локальные и распределенные из хипа объекты не инициализированы
// до момента явной инициализации или присваивания

Account bar()
{
   Account local_acct;
   Account *heap_acct = new Account;
   // ...
```

момент будут содержать случайный набор битов, оставшихся в стеке программы:

```
}
```

Новички часто полагают, что компилятор автоматически генерирует конструктор, если он не задан, и применяет его для инициализации членов класса. Для Account в том виде, в каком мы его определили, это неверно. Никакой конструктор не генерируется и не вызывается. Для более сложных классов, имеющих члены, которые сами являются классами, или использующих наследование, это отчасти справедливо: конструктор по умолчанию может быть сгенерирован, но и он не присваивает начальных значений членам встроенных или составных типов, таким, как указатели или массивы.

Если мы хотим, чтобы подобные члены инициализировались, то должны сами позаботиться об этом, предоставив один или несколько конструкторов. В противном случае отличить корректное значение члена такого типа от неинициализированного, если объект создан локально или распределен из хипа,3 практически невозможно.

14.2.2. Ограничение прав на создание объекта

Доступность конструктора определяется тем, в какой секции класса он объявлен. Мы можем ограничить или явно запретить некоторые формы создания объектов, если поместим соответствующий конструктор в неоткрытую секцию. В примере ниже конструктор по умолчанию класса Account объявлен закрытым, а с двумя параметрами –

```
class Account {
   friend class vector< Account >;
public:
   explicit Account( const char*, double = 0.0 );
   // ...
private:
   Account();
   // ...
```

открытым:

```
};
```

---

3 Для тех, кто раньше программировал на C: приведенное выше определение класса Account на C выглядело бы так:

```
typedef struct {
   char         *_name;
   unsigned int _acct_nmbr;
   double       _balance;
} Account;
```

Обычная программа сможет теперь определять объекты класса Account, лишь указав как имя владельца счета, так и начальный баланс. Однако функции-члены Account и дружественный ему класс vector могут создавать объекты, пользуясь любым конструктором.

Конструкторы, не являющиеся открытыми, в реальных программах C++ чаще всего используются для:

- предотвращения копирования одного объекта в другой объект того же класса (эта проблема рассматривается в следующем подразделе);
- указания на то, что конструктор должен вызываться только в случае, когда данный класс выступает в роли базового в иерархии наследования, а не для создания объектов, которыми программа может манипулировать напрямую (см. обсуждение наследования и объектно-ориентированного программирования в главе 17).

14.2.3. Копирующий конструктор

Инициализация объекта другим объектом того же класса называется *почленной инициализацией по умолчанию*. Копирование одного объекта в другой выполняется путем последовательного копирования каждого нестатического члена. Проектировщик класса может изменить такое поведение, предоставив специальный *копирующий конструктор*. Если он определен, то вызывается всякий раз, когда один объект инициализируется другим объектом того же класса.

Часто почленная инициализация не обеспечивает корректного поведения класса. Поэтому мы явно определяем копирующий конструктор. В нашем классе Account это необходимо, иначе два объекта будут иметь одинаковые номера счетов, что запрещено спецификацией класса.

Копирующий конструктор принимает в качестве формального параметра ссылку на объект класса (традиционно объявляемую со спецификатором const). Вот его

```
inline Account::
Account( const Account &rhs )
       : _balance( rhs._balance )
{
        _name = new char[ strlen(rhs._name) + 1 ];
        strcpy( _name, rhs._name );

        // копировать rhs._acct_nmbr нельзя
        _acct_nmbr = get_unique_acct_nmbr();
```

реализация:

```
}
```

Когда мы пишем:

```
Account acct2( acct1 );
```

компилятор определяет, объявлен ли явный копирующий конструктор для класса Account. Если он объявлен и доступен, то он и вызывается; а если недоступен, то

определение `acct2` считается ошибкой. В случае, когда копирующий конструктор не объявлен, выполняется почленная инициализация по умолчанию. Если впоследствии объявление копирующего конструктора будет добавлено или удалено, никаких изменений в программы пользователей вносить не придется. Однако перекомпилировать их все же необходимо. (Более подробно почленная инициализация рассматривается в разделе 14.6.)

Упражнение 14.1

Какие из следующих утверждений ложны? Почему?

1. У класса должен быть хотя бы один конструктор.
2. Конструктор по умолчанию – это конструктор с пустым списком параметров.
3. Если разумных начальных значений у членов класса нет, то не следует предоставлять конструктор по умолчанию.
4. Если в классе нет конструктора по умолчанию, то компилятор генерирует его автоматически и инициализирует каждый член значением по умолчанию для соответствующего типа.

Упражнение 14.2

Предложите один или несколько конструкторов для данного множества членов.

```
class NoName {
public:
   // здесь должны быть конструкторы
   // ...
protected:
   char   *pstring;
   int    ival;
   double dval;
};
```

Объясните свой выбор:

Упражнение 14.3

Выберите одну из следующих абстракций (или предложите свою собственную). Решите, какие данные (задаваемые пользователем) подходят для представляющего эту абстракцию класса. Напишите соответствующий набор конструкторов. Объясните свое решение.

- Книга
- Дата
- Служащий
- Транспортное средство
- Объект
- Дерево

Упражнение 14.4

Пользуясь приведенным определением класса:

```
class Account {
public:
   Account();
   explicit Account( const char*, double=0.0 );
   // ...
};
```

```
(a) Account acct;
(b) Account acct2 = acct;
(c) Account acct3 = "Rena Stern";
(d) Account acct4( "Anna Engel", 400.00 );
```

объясните, что происходит в результате следующих определений:

```
(e) Account acct5 = Account( acct3 );
```

Упражнение 14.5

Параметр копирующего конструктора может и не быть константным, но обязан быть ссылкой. Почему ошибочна такая инструкция:

```
Account::Account( const Account rhs );
```

14.3. Деструктор класса

Одна из целей, стоящих перед конструктором, – обеспечить автоматическое выделение ресурса. Мы уже видели в примере с классом Account конструктор, где с помощью оператора new выделяется память для массива символов и присваивается уникальный номер счету. Можно также представить ситуацию, когда нужно получить монопольный доступ к разделяемой памяти или к критической секции потока. Для этого необходима симметричная операция, обеспечивающая автоматическое освобождение памяти или возврат ресурса по завершении времени жизни объекта, – *деструктор*. Деструктор – это специальная определяемая пользователем функция-член, которая автоматически вызывается, когда объект выходит из области видимости или когда к указателю на объект применяется операция delete. Имя этой функции образовано из имени класса с предшествующим символом “тильда” (~). Деструктор не возвращает значения и не принимает никаких параметров, а следовательно, не может быть перегружен. Хотя разрешается определять несколько таких функций-членов, лишь одна из них будет применяться ко всем объектам класса. Вот, например, деструктор для нашего класса Account:

```
class Account {
public:
          Account();
     explicit Account( const char*, double=0.0 );
          Account( const Account& );
     ~Account();
     // ...
private:
          char         *_name;
          unsigned int _acct_nmbr;
          double       _balance;
};

inline
Account::~Account()
{
      delete [] _name;
      return_acct_number( _acct_nnmbr );
}
```

```
inline
Account::~Account()
{
      // необходимо
      delete [] _name;
      return_acct_number( _acct_nnmbr );
      // необязательно
      _name = 0;
      _balance = 0.0;
      _acct_nmbr = 0;
}
```

Обратите внимание, что в нашем деструкторе не сбрасываются значения членов:

Делать это необязательно, поскольку отведенная под члены объекта память все равно

```
class Point3d {
public:
   // ...
private:
   float x, y, z;
};
```

будет освобождена. Рассмотрим следующий класс:

Конструктор здесь необходим для инициализации членов, представляющих координаты точки. Нужен ли деструктор? Нет. Для объекта класса `Point3d` не требуется освобождать ресурсы: память выделяется и освобождается компилятором автоматически в начале и в конце его жизни.

В общем случае, если члены класса имеют простые значения, скажем, координаты точки, то деструктор не нужен. Не для каждого класса необходим деструктор, даже если у него есть один или более конструкторов. Основной целью деструктора является освобождения

ресурсов, выделенных либо в конструкторе, либо во время жизни объекта, например освобождение замка или памяти, выделенной оператором new.

Но функции деструктора не ограничены только освобождением ресурсов. Он может реализовывать любую операцию, которая по замыслу проектировщика класса должна быть выполнена сразу по окончании использования объекта. Так, широко распространенным приемом для измерения производительности программы является определение класса Timer, в конструкторе которого запускается та или иная форма программного таймера. Деструктор останавливает таймер и выводит результаты замеров. Объект данного класса можно условно определять в критических участках программы,

```
{
   // начало критического участка программы
#ifdef PROFILE
   Timer t;
#endif
   // критический участок
   // t уничтожается автоматически
   // отображается затраченное время ...
}
```

которые мы хотим профилировать, таким образом:

Чтобы убедиться в том, что мы понимаем поведение деструктора (да и конструктора

```
(1)    #include "Account.h"
(2)    Account global( "James Joyce" );
(3)    int main()
(4)    {
(5)       Account local( "Anna Livia Plurabelle", 10000 );
(6)       Account &loc_ref = global;
(7)       Account *pact = 0;
(8)
(9)       {
(10)          Account local_too( "Stephen Hero" );
(11)          pact = new Account( "Stephen Dedalus" );
(12)      }
(13)
(14)      delete pact;
(15)   }
```

тоже), разберем следующий пример:

Сколько здесь вызывается конструкторов? Четыре: один для глобального объекта global в строке (2); по одному для каждого из локальных объектов local и local_too в строках (5) и (10) соответственно, и один для объекта, распределенного в хипе, в строке (11). Ни объявление ссылки loc_ref на объект в строке (6), ни объявление указателя pact в строке (7) не приводят к вызову конструктора. Ссылка – это псевдоним для уже сконструированного объекта, в данном случае для global. Указатель также лишь адресует объект, созданный ранее (в данном случае распределенный в хипе, строка (11)), или не адресует никакого объекта (строка (7)).

Аналогично вызываются четыре деструктора: для глобального объекта global, объявленного в строке (2), для двух локальных объектов и для объекта в хипе при вызове delete в строке (14). Однако в программе нет инструкции, с которой можно связать

вызов деструктора. Компилятор просто вставляет эти вызовы за последним использованием объекта, но перед закрытием соответствующей области видимости.

Конструкторы и деструкторы глобальных объектов вызываются на стадиях инициализации и завершения выполнения программы. Хотя такие объекты нормально ведут себя при использовании в том файле, где они определены, но их применение в ситуации, когда производятся ссылки через границы файлов, становится в C++ серьезной проблемой.4

Деструктор не вызывается, когда из области видимости выходит ссылка или указатель на объект (сам объект при этом остается).

C++ с помощью внутренних механизмов препятствует применению оператора `delete` к указателю, не адресующему никакого объекта, так что соответствующие проверки кода

```
// необязательно: неявно выполняется компилятором
```

необязательны:

```
if (pact != 0 ) delete pact;
```

Всякий раз, когда внутри функции этот оператор применяется к отдельному объекту, размещенному в хипе, лучше использовать объект класса `auto_ptr`, а не обычный указатель (см. обсуждение класса `auto_ptr` в разделе 8.4). Это особенно важно потому, что пропущенный вызов `delete` (скажем, в случае, когда возбуждается исключение) ведет не только к утечке памяти, но и к пропуску вызова деструктора. Ниже приводится пример программы, переписанной с использованием `auto_ptr` (она слегка модифицирована, так как объект класса `auto_ptr` может быть явно переустановлен для

```
#include <memory>
#include "Account.h"
Account global( "James Joyce" );
int main()
{
   Account local( "Anna Livia Plurabelle", 10000 );
   Account &loc_ref = global;
   auto_ptr<Account> pact( new Account( "Stephen Dedalus" ));

   {
      Account local_too( "Stephen Hero" );
   }

   // объект auto_ptr уничтожается здесь
```

адресации другого объекта только присваиванием его другому `auto_ptr`):

```
}
```

4 См. статью Джерри Шварца в [LIPPMAN96b], где приводится дискуссия по этому поводу и описывается решение, остающееся пока наиболее распространенным.

14.3.1. Явный вызов деструктора

Иногда вызывать деструктор для некоторого объекта приходится явно. Особенно часто такая необходимость возникает в связи с оператором new (см. раздел 8.4). Рассмотрим пример. Когда мы пишем:

```
char *arena = new char[ sizeof Image ];
```

то из хипа выделяется память, размер которой равен размеру объекта типа Image, она не инициализирована и заполнена случайными битами. Если же написать:

```
Image *ptr = new (arena) Image( "Quasimodo" );
```

то никакой новой памяти не выделяется. Вместо этого переменной ptr присваивается адрес, ассоциированный с переменной arena. Теперь память, на которую указывает ptr, интерпретируется как занимаемая объектом класса Image, и конструктор применяется к уже существующей области. Таким образом, оператор размещения new() позволяет сконструировать объект в ранее выделенной области памяти.

Закончив работать с изображением Quasimodo, мы можем произвести какие-то операции с изображением Esmerelda, размещенным по тому же адресу arena в памяти:

```
Image *ptr = new (arena) Image( "Esmerelda" );
```

Однако изображение Quasimodo при этом будет затерто, а мы его модифицировали и хотели бы записать на диск. Обычно сохранение выполняется в деструкторе класса

```
// плохо: не только вызывает деструктор, но и освобождает память
```

Image, но если мы применим оператор delete:

```
delete ptr;
```

то, помимо вызова деструктора, еще и возвратим в хип память, чего делать не следовало бы. Вместо этого можно явно вызвать деструктор класса Image:

```
ptr->~Image();
```

сохранив отведенную под изображение память для последующего вызова оператора размещения new.

Отметим, что, хотя ptr и arena адресуют одну и ту же область памяти в хипе,

```
// деструктор не вызывается
```

применение оператора delete к arena

```
delete arena;
```

не приводит к вызову деструктора класса Image, так как arena имеет тип char*, а компилятор вызывает деструктор только тогда, когда операндом в delete является указатель на объект класса, имеющего деструктор.

14.3.2. Опасность увеличения размера программы

Встроенный деструктор может стать причиной непредвиденного увеличения размера программы, поскольку он вставляется в каждой точке выхода внутри функции для

```
Account acct( "Tina Lee" );
int swt;
// ...
switch( swt ) {
case 0:
   return;
case 1:
   // что-то сделать
   return;
case 2:
   // сделать что-то другое
   return;
// и так далее
}
```

каждого активного локального объекта. Например, в следующем фрагменте

компилятор подставит деструктор перед каждой инструкцией `return`. Деструктор класса `Account` невелик, и затраты времени и памяти на его подстановку тоже малы. В противном случае придется либо объявить деструктор невстроенным, либо реорганизовать программу. В примере выше инструкцию `return` в каждой метке `case` можно заменить инструкцией `break` с тем, чтобы у функции была единственная точка

```
// переписано для обеспечения единственной точки выхода
switch( swt ) {
case 0:
   break;
case 1:
   // что-то сделать
   break;
case 2:
   // сделать что-то другое
   break;
// и так далее
}

// единственная точка выхода
```

выхода:

```
return;
```

Упражнение 14.6

Напишите подходящий деструктор для приведенного набора членов класса, среди которых `pstring` адресует динамически выделенный массив символов:

```
class NoName {
public:
   ~NoName();
   // ...
private:
   char    *pstring;
   int     ival;
   double  dval;
};
```

Упражнение 14.7

Необходим ли деструктор для класса, который вы выбрали в упражнении 14.3? Если нет, объясните почему. В противном случае предложите реализацию.

Упражнение 14.8

```
void mumble( const char *name, fouble balance, char acct_type )
{
   Account acct;

   if ( ! name )
      return;

   if ( balance <= 99 )
      return;

   switch( acct_type ) {
      case 'z': return;
      case 'a':
      case 'b': return;
   }

   // ...
}
```

Сколько раз вызываются деструкторы в следующем фрагменте:

14.4. Массивы и векторы объектов

Массив объектов класса определяется точно так же, как массив элементов встроенного типа. Например:

```
Account table[ 16 ];
```

определяет массив из 16 объектов `Account`. Каждый элемент по очереди инициализируется конструктором по умолчанию. Можно и явно передать конструкторам аргументы внутри заключенного в фигурные скобки списка инициализации массива. Строка:

```
Account pooh_pals[] = { "Piglet", "Eeyore", "Tigger" };
```

определяет массив из трех элементов, инициализируемых конструкторами:

```
Account( "Piglet", 0.0 );   // первый элемент (Пятачок)
Account( "Eeyore", 0.0 );   // второй элемент (Иа-Иа)

Account( "Tigger", 0.0 );   // третий элемент (Тигра)
```

Один аргумент можно задать явно, как в примере выше. Если же необходимо передать

```
Account pooh_pals[] = {
   Account( "Piglet", 1000.0 ),
   Account( "Eeyore", 1000.0 ),
   Account( "Tigger", 1000.0 )
};
```

несколько аргументов, то придется воспользоваться явным вызовом конструктора:

Чтобы включить в список инициализации массива конструктор по умолчанию, мы

```
Account pooh_pals[] = {
   Account( "Woozle", 10.0 ),        // Бука
   Account( "Heffalump", 10.0 ),     // Слонопотам
   Account();
};
```

употребляем явный вызов с пустым списком параметров:

```
Account pooh_pals[3] = {
   Account( "Woozle", 10.0 ),
   Account( "Heffalump", 10.0 )
};
```

Эквивалентный массив из трех элементов можно объявить и так:

Таким образом, члены списка инициализации последовательно используются для заполнения очередного элемента массива. Те элементы, для которых явные аргументы не заданы, инициализируются конструктором по умолчанию. Если его нет, то в списке должны быть заданы аргументы конструктора для каждого элемента массива.

Доступ к отдельным элементам массива объектов производится с помощью оператора взятия индекса, как и для массива элементов любого из встроенных типов. Например:

```
pooh_pals[0];
```

обращается к `Piglet`, а

```
pooh_pals[1];
```

к `Eeyore` и т.д. Для доступа к членам объекта, находящегося в некотором элементе массива, мы сочетаем операторы взятия индекса и доступа к членам:

```
pooh_pals[1]._name != pooh_pals[2]._name;
```

Не существует способа явно указать начальные значения элементов массива, память для которого выделена из хипа. Если класс поддерживает создание динамических массивов с помощью оператора new, он должен либо иметь конструктор по умолчанию, либо не иметь никаких конструкторов. На практике почти у всех классов есть такой конструктор.

Объявление

```
Account *pact = new Account[ 10 ];
```

создает в памяти, выделенной из хипа, массив из десяти объектов класса Account, причем каждый инициализируется конструктором по умолчанию.

Чтобы уничтожить массив, адресованный указателем pact, необходимо применить

```
// увы! это не совсем правильно
```

оператор delete. Однако написать

```
delete pact;
```

недостаточно, так как pact при этом не идентифицируется как массив объектов. В результате деструктор класса Account применяется лишь к первому элементу массива. Чтобы применить его к каждому элементу, мы должны включить пустую пару скобок

```
// правильно:
// показывает, что pact адресует массив
```

между оператором delete и адресом удаляемого объекта:

```
delete [] pact;
```

Пустая пара скобок говорит о том, что pact адресует именно массив. Компилятор определяет, сколько в нем элементов, и применяет деструктор к каждому из них.

14.4.1. Инициализация массива, распределенного из хипа A

По умолчанию инициализация массива объектов, распределенного из хипа, проходит в два этапа: выделение памяти для массива, к каждому элементу которого применяется конструктор по умолчанию, если он определен, и последующее присваивание значения каждому элементу.

Чтобы свести инициализацию к одному шагу, программист должен вмешаться и поддержать следующую семантику: задать начальные значения для всех или некоторых элементов массива и гарантировать применение конструктора по умолчанию для тех элементов, начальные значения которых не заданы. Ниже приведено одно из возможных программных решений, где используется оператор размещения new:

```
#include <utility>
#include <vector >
#include <new>
#include <cstddef>
#include "Accounts.h"

typedef pair<char*, double> value_pair;

/* init_heap_array()
 *     объявлена как статическая функция-член
 *     обеспечивает выделение памяти из хипа и инициализацию
 *     массива объектов
 * init_values: пары начальных значений элементов массива
 * elem_count: число элементов в массиве
 *             если 0, то размером массива считается размер вектора
 *             init_values
 */
Account*
Account::
init_heap_array(
        vector<value_pair> &init_values,
        vector<value_pair>::size_type elem_count = 0 )
{
   vector<value_pair>::size_type
          vec_size = init_value.size();

   if ( vec_size == 0 && elem_count == 0 )
      return 0;

   // размер массива равен либо elem_count,
   // либо, если elem_count == 0, размеру вектора ...
   size_t elems = elem_count
                  ? elem_count : vec_size();

   // получить блок памяти для размещения массива
   char *p = new char[sizeof(Account)*elems];

   // по отдельности инициализировать каждый элемент массива
   int offset = sizeof( Account );
   for ( int ix = 0; ix < elems; ++ix )
   {
        // смещение ix-ого элемента
        // если пара начальных значений задана,
        //      передать ее конструктору;
        // в противном случае вызвать конструктор по умолчанию

        if ( ix < vec_size )
           new( p+offset*ix ) Account( init_values[ix].first,
                                       init_values[ix].second );
        else new( p+offset*ix ) Account;
   }

   // отлично: элементы распределены и инициализированы;
   // вернуть указатель на первый элемент
   return (Account*)p;
}
```

Необходимо заранее выделить блок памяти, достаточный для хранения запрошенного массива, как массив байт, чтобы избежать применения к каждому элементу конструктора по умолчанию. Это делается в такой инструкции:

```
char *p = new char[sizeof(Account)*elems];
```

Далее программа в цикле обходит этот блок, присваивая на каждой итерации переменной p адрес следующего элемента и вызывая либо конструктор с двумя параметрами, если

```
for ( int ix = 0; ix < elems; ++ix )
   {
       if ( ix < vec_size )
           new( p+offset*ix ) Account( init_values[ix].first,
                                       init_values[ix].second );
        else new( p+offset*ix ) Account;
```

задана пара начальных значений, либо конструктор по умолчанию:

```
   }
```

В разделе 14.3 говорилось, что оператор размещения new позволяет применить конструктор класса к уже выделенной области памяти. В данном случае мы используем new для поочередного применения конструктора класса Account к каждому из выделенных элементов массива. Поскольку при создании инициализированного массива мы подменили стандартный механизм выделения памяти, то должны сами позаботиться о ее освобождении. Оператор delete работать не будет:

```
delete [] ps;
```

Почему? Потому что ps (мы предполагаем, что эта переменная была инициализирована вызовом init_heap_array()) указывает на блок памяти, полученный не с помощью стандартного оператора new, поэтому число элементов в массиве компилятору

```
void
Account::
dealloc_heap_array( Account *ps, size_t elems )
{
   for ( int ix = 0; ix < elems; ++ix )
       ps[ix].Account::~Account();

   delete [] reinterpret_cast<char*>(ps);
```

неизвестно. Так что всю работу придется сделать самим:

```
}
```

Если в функции инициализации мы пользовались арифметическими операциями над указателями для доступа к элементам:

```
new( p+offset*ix ) Account;
```

то здесь мы обращаемся к ним, задавая индекс в массиве ps:

```
ps[ix].Account::~Account();
```

Хотя и ps, и p адресуют одну и ту же область памяти, ps объявлен как указатель на объект класса Account, а p – как указатель на char. Индексирование p дало бы ix-й байт, а не ix-й объект класса Account. Поскольку с p ассоциирован не тот тип, что нужно,

арифметические операции над указателями приходится программировать самостоятельно.

Мы объявляем обе функции статическими членами класса:

```
class Account {
public:
     // ...
          static Account* init_heap_array(
          vector<value_pair> &init_values,
          vector<value_pair>::size_type elem_count = 0 );
          static void dealloc_heap_array( Account*, size_t );
     // ...

typedef pair<char*, double> value_pair;
};
```

14.4.2. Вектор объектов

Когда определяется вектор из пяти объектов класса, например:

```
vector< Point > vec( 5 );
```

то инициализация элементов производится в следующем порядке5:

1. С помощью конструктора по умолчанию создается временный объект типа класса, хранящегося в векторе. .
2. К каждому элементу вектора применяется копирующий конструктор, в результате чего каждый объект инициализируется копией временного объекта.
3. Временный объект уничтожается.

Хотя конечный результат оказывается таким же, как при определении массива из пяти объектов класса:

```
Point pa[ 5 ];
```

эффективность подобной инициализации вектора ниже, так как, во-первых, на конструирование и уничтожение временного объекта, естественно, нужны ресурсы, а во-вторых, копирующий конструктор обычно оказывается вычислительно более сложным, чем конструктор по умолчанию.

Общее правило проектирования таково: вектор объектов класса удобнее только для вставки элементов, т.е. в случае, когда изначально определяется пустой вектор. Если мы заранее вычислили, сколько придется вставлять элементов, или имеем на этот счет обоснованное предположение, то надо зарезервировать необходимую память, а затем приступать к вставке. Например:

---

5 Сигнатура ассоциированного конструктора имеет следующий смысл. Копирующий конструктор применяет некоторое значение к каждому элементу по очереди. Задавая в качестве второго аргумента объект класса, мы делаем создание временного объекта излишним:

```
explicit vector( size_type n, const T& value=T(), const Allocator&=Allocator());
```

```
vector< Point > cvs;   // пустой
int cv_cnt = calc_control_vertices();

// зарезервировать память для хранения cv_cnt объектов класса Point
// cvs все еще пуст ...
cvs.reserve( cv_cnt );
// открыть файл и подготовиться к чтению из него
ifstream infile( "spriteModel" );
istream_iterator<Point> cvfile( infile ),eos;

// вот теперь можно вставлять элементы
copy( cvfile, eos, inserter( cvs, cvs.begin() ));
```

(Алгоритм `copy()`, итератор вставки `inserter` и потоковый итератор чтения `istream_iterator` рассматривались в главе 12.) Поведение объектов `list` (список) и `deque` (двусторонняя очередь) аналогично поведению объектов `vector` (векторов). Вставка объекта в любой из этих контейнеров осуществляется с помощью копирующего конструктора.

Упражнение 14.9

Какие из приведенных инструкций неверны? Исправьте их.

```
(b) Account iA[1024] = {
    "Nhi", "Le", "Jon", "Mike", "Greg", "Brent", "Hank"
```

```
(a) Account *parray[10] = new Account[10];

    "Roy", "Elena" };
```

```
(d) string as[] = *ps;
```

```
(c) string *ps=string[5]("Tina","Tim","Chyuan","Mira","Mike");
```

Упражнение 14.10

Что лучше применить в каждой из следующих ситуаций: статический массив (такой, как `Account pA[10]`), динамический массив или вектор? Объясните свой выбор.

Внутри функции `Lut()` нужен набор из `256` элементов для хранения объектов класса `Color`. Значения являются константами.

Необходимо хранить набор из неизвестного числа объектов класса `Account`. Данные счетов читаются из файла.

Функция `gen_words(elem_size)` должна сгенерировать и передать обработчику текста набор из `elem_size` строк.

Упражнение 14.11

Потенциальным источником ошибок при использовании динамических массивов является пропуск пары квадратных скобок, говорящей, что указатель адресует массив, т.е.

```
// печально: не проверяется, что parray адресует массив
```

неверная запись

```
delete parray;
```

```
// правильно: определяется размер массива, адресуемого parray
```

вместо

```
delete [] parray;
```

Наличие пары скобок заставляет компилятор найти размер массива. Затем к каждому элементу по очереди применяется деструктор (всего `size` раз). Если же скобок нет, уничтожается только один элемент. В любом случае освобождается вся память, занятая массивом.

При обсуждении первоначального варианта языка C++ много спорили о том, должно ли наличие квадратных скобок инициировать поиск или же (как было в исходной

```
// в первоначальном варианте языка размер массива требовалось задавать
          явно
```

спецификации) лучше поручить программисту явно указывать размер массива:

```
delete p[10] parray;
```

Как вы думаете, почему язык был изменен таким образом, что явного задания размера не требуется (а значит, нужно уметь его сохранять и извлекать), но скобки, хотя и пустые, в операторе `delete` остались (так что компилятор не должен запоминать, адресует указатель единственный объект или массив)? Какой вариант языка предложили бы вы?

14.5. Список инициализации членов

```
#include <string>
class Account {
public:
   // ...
private:
   unsigned int _acct_nmbr;
   double       _balance;
   string       _name;
```

Модифицируем наш класс `Account`, объявив член `_name` типа `string`:

```
};
```

Придется заодно изменить и конструкторы. Возникает две проблемы: поддержание совместимости с первоначальным интерфейсом и инициализация объекта класса с помощью подходящего набора конструкторов.

Исходный конструктор Account с двумя параметрами

```
Account( const char*, double = 0.0 );
```

```
string new_client( "Steve Hall" );
```

не может инициализировать член типа string. Например:

```
Account new_acct( new_client, 25000 );
```

не будет компилироваться, так как не существует неявного преобразования из типа string в тип char*. Инструкция

```
Account new_acct( new_client.c_str(), 25000 );
```

правильна, но вызовет у пользователей класса недоумение. Одно из решений – добавить новый конструктор вида:

```
Account( string, double = 0.0 );
```

Если написать:

```
Account new_acct( new_client, 25000 );
```

```
Account *open_new_account( const char *nm )
{
   Account *pact = new Account( nm );
   // ...
   return pacct;
```

вызывается именно этот конструктор, тогда как старый код

```
}
```

по-прежнему будет приводить к вызову исходного конструктора с двумя параметрами.

Так как в классе string определено преобразование из типа char* в тип string (преобразования классов обсуждаются в этой главе ниже), то можно заменить исходный конструктор на новый, которому в качестве первого параметра передается тип string. В таком случае, когда встречается инструкция:

```
Account myAcct( "Tinkerbell" );
```

"Tinkerbell" преобразуется во временный объект типа string. Затем этот объект передается новому конструктору с двумя параметрами.

При проектировании приходится идти на компромисс между увеличением числа конструкторов класса Account и несколько менее эффективной обработкой аргументов

типа char* из-за необходимости создавать временный объект. Мы предоставили две версии конструктора с двумя параметрами. Тогда модифицированный набор

```
#include <string>

class Account {
public:
   Account();
   Account( const char*, double=0.0 );
   Account( const string&, double=0.0 );
   Account( const Account& );
   // ...
private:
   // ...
```

конструкторов Account будет таким:

```
};
```

Как правильно инициализировать член, являющийся объектом некоторого класса с собственным набором конструкторов? Этот вопрос можно разделить на три:

1. где вызывается конструктор по умолчанию? Внутри конструктора по умолчанию класса Account;
2. где вызывается копирующий конструктор? Внутри копирующего конструктора класса Account и внутри конструктора с двумя параметрами, принимающего в качестве первого тип string;
3. как передать аргументы конструктору класса, являющегося членом другого класса? Это необходимо делать внутри конструктора Account с двумя параметрами, принимающего в качестве первого тип char*.

Решение заключается в использовании списка инициализации членов (мы упоминали о нем в разделе 14.2). Члены, являющиеся классами, можно явно инициализировать с помощью списка, состоящего из разделенных запятыми пар “имя члена/значение”. Наш конструктор с двумя параметрами теперь выглядит так (напомним, что _name – это член,

```
inline Account::
Account( const char* name, double opening_bal )
       : _name( name ), _balance( opening_bal )
{
       _acct_nmbr = het_unique_acct_nmbr();
```

являющийся объектом класса string):

```
}
```

Список инициализации членов следует за сигнатурой конструктора и отделяется от нее двоеточием. В нем указывается имя члена, а в скобках – начальные значения, что аналогично синтаксису вызова функции. Если член является объектом класса, то эти значения становятся аргументами, передаваемыми подходящему конструктору, который затем и используется. В нашем примере значение name передается конструктору string, который применяется к члену _name. Член _balance инициализируется значением opening_bal.

Аналогично выглядит второй конструктор с двумя параметрами:

```
inline Account::
Account( const string& name, double opening_bal )
       : _name( name ), _balance( opening_bal )
{
       _acct_nmbr = het_unique_acct_nmbr();
}
```

В этом случае вызывается копирующий конструктор `string`, инициализирующий член `_name` значением параметра `name` типа `string`.

Часто у новичков возникает вопрос: в чем разница между использованием списка инициализации и присваиванием значений членам в теле конструктора? Например, в чем

```
inline Account::
Account( const char* name, double opening_bal )
       : _name( name ), _balance( opening_bal )
{
       _acct_nmbr = het_unique_acct_nmbr();
}
```

разница между

```
Account( const char* name, double opening_bal )
{
       _name = name;
       _balance = opening_bal;
       _acct_nmbr = het_unique_acct_nmbr();
}
```

и

В конце работы обоих конструкторов все три члена будут иметь одинаковые значения. Разница в том, что только список обеспечивает инициализацию тех членов, которые являются объектами класса. В теле конструктора установка значения члена – это не инициализация, а присваивание. Важно это различие или нет, зависит от природы члена.

С концептуальной точки зрения выполнение конструктора состоит из двух фаз: фаза явной или неявной инициализации и фаза вычислений, включающая все инструкции в теле конструктора. Любая установка значений членов во второй фазе рассматривается как присваивание, а не инициализация. Непонимание этого различия приводит к ошибкам и неэффективным программам.

Первая фаза может быть явной или неявной в зависимости от того, имеется ли список инициализации членов. При неявной инициализации сначала вызываются конструкторы по умолчанию всех базовых классов в порядке их объявления, а затем конструкторы по умолчанию всех членов, являющихся объектами классов. (Базовые классы мы будем рассматривать в главе 17 при обсуждении объектно-ориентированного программирования.) Например, если написать:

```
inline Account::
Account()
{
   _name = "";
   _balance = 0.0;
   _acct_nmbr = 0;
}
```

то фаза инициализации будет неявной. Еще до выполнения тела конструктора вызывается конструктор по умолчанию класса string, ассоциированный с членом _name. Это означает, что присваивание _name пустой строки излишне.

Для объектов классов различие между инициализацией и присваиванием существенно. Член, являющийся объектом класса, всегда следует инициализировать с помощью списка, а не присваивать ему значение в теле конструктора. Более правильной является

```
inline Account::
Account() : _name( string() )
{
   _balance = 0.0;
   _acct_nmbr = 0;
}
```

следующая реализация конструктора по умолчанию класса Account:

Мы удалили ненужное присваивание _name из тела конструктора. Явный же вызов конструктора по умолчанию string излишен. Ниже приведена эквивалентная, но более

```
inline Account::
Account()
{
   _balance = 0.0;
   _acct_nmbr = 0;
}
```

компактная версия:

Однако мы еще не ответили на вопрос об инициализации двух членов встроенных типов. Например, так ли существенно, где происходит инициализация _balance: в списке инициализации или в теле конструктора? Инициализация и присваивание членам, не являющимся объектами классов, эквивалентны как с точки зрения результата, так и с точки зрения производительности (за двумя исключениями). Мы предпочитаем

```
// предпочтительный стиль инициализации
inline Account::
Account() : _balance( 0.0 ), _acct_nmbr( 0 )
```

использовать список:

```
{}
```

Два вышеупомянутых исключения – это константные члены и члены-ссылки независимо от типа. Для них всегда нужно использовать список инициализации, в противном случае

```
class ConstRef {
public:
   ConstRef(int ii );
private:
   int i;
   const int ci;
   int &ri;
};

ConstRef::
ConstRef( int ii )
{  // присваивание
   i = ii;        // правильно
   ci = ii;       // ошибка: нельзя присваивать константному члену
   ri = i;        // ошибка: ri не инициализирована
}
```

компилятор выдаст ошибку:

К началу выполнения тела конструктора инициализация всех константных членов и членов-ссылок должна быть завершена. Для этого нужно указать их в списке

```
// правильно: инициализируются константные члены и ссылки
ConstRef::
ConstRef( int ii )
        : ci( ii ), ri ( i )
{ i = ii; }
```

инициализации. Правильная реализация предыдущего примера такова:

Каждый член должен встречаться в списке инициализации не более одного раза. Порядок инициализации определяется не порядком следования имен в списке, а порядком

```
class Account {
public:
   // ...
private:
   unsigned int _acct_nmbr;
   double       _balance;
   string       _name;
};
```

объявления членов. Если дано следующее объявление членов класса Account:

```
inline Account::
Account() : _name( string() ), _balance( 0.0 ), _acct_nmbr( 0 )
{}
```

то порядок инициализации для такой реализации конструктора по умолчанию

будет следующим: _acct_nmbr, _balance, _name. Однако члены, указанные в списке (или в неявно инициализируемом члене-объекте класса), всегда инициализируются раньше, чем производится присваивание членам в теле конструктора. Например, в

```
inline Account::
Account( const char* name, double bal )
       : _name( name ), _balance( bal )
{
       _acct_nmbr = get_unique_acct_nmbr();
```

следующем конструкторе:

```
}
```

порядок инициализации такой: _balance, _name, _acct_nmbr.

Расхождение между порядком инициализации и порядком следования членов в соответствующем списке может приводить к трудным для обнаружения ошибкам, когда

```
class X {
   int i;
   int j;
public:
   // видите проблему?
   X( int val )
      : j( val ), i( j )
      {}
   // ...
```

один член класса используется для инициализации другого:

```
};
```

Кажется, что перед использованием для инициализации i член j уже инициализирован значением val, но на самом деле i инициализируется первым, для чего применяется еще неинициализированный член j. Мы рекомендуем помещать инициализацию одного

```
// предпочтительная идиома
```

члена другим (если вы считаете это необходимым) в тело конструктора:

```
X::X( int val ) : i( val ) { j = i; }
```

Упражнение 14.12

Что неверно в следующих определениях конструкторов? Как бы вы исправили обнаруженные ошибки?

```
(a) Word::Word( char *ps, int count = 1 )
         : _ps( new char[strlen(ps)+1] ),
           _count( count )
    {
         if ( ps )
            strcpy( _ps, ps );
         else {
            _ps = 0;
            _count = 0;
         }
```

```
(b) class CL1 {
    public:
       CL1() { c.real(0.0); c.imag(0.0); s = "not set"; }
       // ...
    private:
       complex<double> c;
       string s;
```

```
 (c) class CL2 {
    public:
       CL2( map<string,location> *pmap, string key )
            : _text( key ), _loc( (*pmap)[key] ) {}
       // ...
    private:
       location _loc;
       string   _text;
```

```
Account oldAcct( "Anna Livia Plurabelle" );
```

```
    }

    }

};
```

14.6. Почленная инициализация A

Инициализация одного объекта класса другим объектом того же класса, как, например:

```
Account newAcct( oldAcct );
```

называется *почленной инициализацией по умолчанию*. По умолчанию – потому, что она производится автоматически, независимо от того, есть явный конструктор или нет. Почленной – потому, что единицей инициализации является отдельный нестатический член, а не побитовая копия всего объекта класса.

Такую инициализацию проще всего представить, если считать, что компилятор создает специальный внутренний копирующий конструктор, где поочередно, в порядке объявления, инициализируются все нестатические члены. Если рассмотреть первое

```
class Account {
public:
   // ...
private:
   char         *_name;
   unsigned int _acct_nmbr;
   double       _balance;
```

определение нашего класса Account:

```
};
```

```
inline Account::
Account( const Account &rhs )
{
   _name = rhs._name;
   _acct_nmbr = rhs._acct_nmbr;
   _balance = rhs._balance;
```

то можно представить, что копирующий конструктор по умолчанию определен так:

```
}
```

Почленная инициализация одного объекта класса другим встречается в следующих ситуациях:

- явная инициализация одного объекта другим:

```
Account newAcct( oldAcct );
```

```
extern bool cash_on_hand( Account acct );

if ( cash_on_hand( oldAcct ))
```

- передача объекта класса в качестве аргумента функции:

```
   // ...
```

- передача объекта класса в качестве возвращаемого функцией значения:

```
extern Account
       consolidate_accts( const vector< Account >& )
{
   Account final_acct;
   // выполнить финансовую операцию
   return final_acct;
}


// вызывается пять копирующих конструкторов класса string
```

- определение непустого последовательного контейнера:

```
vector< string > svec( 5 );
```

(В этом примере с помощью конструктора `string` по умолчанию создается один временный объект, который затем копируется в пять элементов вектора посредством копирующего конструктора `string`.)

- вставка объекта класса в контейнер:

```
svec.push_back( string( "pooh" ));
```

Для большинства определений реальных классов почленная инициализация по умолчанию не соответствует семантике класса. Чаще всего это случается, когда его член представляет собой указатель, который адресует освобождаемую деструктором память в хипе, как, например, в нашем `Account`.

В результате такой инициализации `newAcct._name` и `oldAcct._name` указывают на одну и ту же С-строку. Если `oldAcct` выходит из области видимости и к нему применяется деструктор, то `newAcct._name` указывает на освобожденную область памяти. С другой стороны, если `newAcct` модифицирует строку, адресуемую `_name`, то она изменяется и для `oldAcct`. Подобные ошибки очень трудно найти.

Одно из решений псевдонимов указателей заключается в том, чтобы выделить область памяти для копии строки и инициализировать `newAcct._name` адресом этой области. Следовательно, почленную инициализацию по умолчанию для класса `Account` нужно подавить за счет предоставления явного копирующего конструктора, который реализует правильную семантику инициализации.

Внутренняя семантика класса также может не соответствовать почленной инициализации по умолчанию. Ранее мы уже объясняли, что два разных объекта `Account` не должны иметь одинаковые номера счетов. Чтобы гарантировать такое поведение, мы должны подавить почленную инициализацию по умолчанию для класса `Account`. Вот как выглядит копирующий конструктор, решающий обе эти проблемы:

```
inline Account::
Account( const Account &rhs )
{
   // решить проблему псевдонима указателя
   _name = new char[ strlen(rhs._name)+1 ];
   strcpy( _name, rhs._name );

   // решить проблему уникальности номера счета
   _acct_nmbr = get_unique_acct_nmbr();

   // копирование этого члена и так работает
   _balance = rhs._balance;
}
```

Альтернативой написанию копирующего конструктора является полный запрет почленной инициализации. Это можно сделать следующим образом:

1. Объявить копирующий конструктор закрытым членом. Это предотвратит почленную инициализацию всюду, кроме функций-членов и друзей класса.
2. Запретить почленную инициализацию в функциях-членах и друзьях класса, намеренно не предоставляя определения копирующего конструктора (однако объявить его так, как описано на шаге 1, все равно нужно). Язык не дает нам возможности ограничить доступ к закрытым членам класса со стороны функций-членов и друзей. Но если определение отсутствует, то любая попытка вызвать копирующий конструктор, законная с точки зрения компилятора, приведет к ошибке во время редактирования связей, поскольку не удастся найти определение символа.

```
class Account {
public:
   Account();
   Account( const char*, double=0.0 );

   // ...
private:
   Account( const Account& );
   // ...
};
```

Чтобы запретить почленную инициализацию, класс `Account` можно объявить так:

14.6.1. Инициализация члена, являющегося объектом класса

Что произойдет, если в объявлении `_name` заменить С-строку на тип класса `string`? Как это повлияет на почленную инициализацию по умолчанию? Как надо будет изменить явный копирующий конструктор? Мы ответим на эти вопросы в данном подразделе.

При почленной инициализации по умолчанию исследуется каждый член. Если он принадлежит к встроенному или составному типу, то такая инициализация применяется непосредственно. Например, в первоначальном определении класса `Account` член `_name` инициализируется непосредственно, так как это указатель:

```
newAcct._name = oldAcct._name;
```

Члены, являющиеся объектами классов, обрабатываются по-другому. В инструкции

```
Account newAcct( oldAcct );
```

оба объекта распознаются как экземпляры Account. Если у этого класса есть явный копирующий конструктор, то он и применяется для задания начального значения, в противном случае выполняется почленная инициализация по умолчанию.

Таким образом, если обнаруживается член-объект класса, то описанный выше процесс применяется рекурсивно. У класса есть явный копирующий конструктор? Если да, вызвать его для задания начального значения члена-объекта класса. Иначе применить к этому члену почленную инициализацию по умолчанию. Если все члены этого класса принадлежат к встроенным или составным типам, то каждый инициализируется непосредственно и процесс на этом завершается. Если же некоторые члены сами являются объектами классов, то алгоритм применяется к ним рекурсивно, пока не останется ничего, кроме встроенных и составных типов.

В нашем примере у класса string есть явный копирующий конструктор, поэтому _name инициализируется с помощью его вызова. Копирующий конструктор по умолчанию для

```
inline Account::
Account( const Account &rhs )
{
   _acct_nmbr = rhs._acct_nmbr;
   _balance = rhs._balance;

   // Псевдокод на C++
   // иллюстрирует вызов копирующего конструктора
   // для члена, являющегося объектом класса
   _name.string::string( rhs._name );
}
```

класса Account выглядит следующим образом (хотя явно он не определен):

```
// не совсем правильно...
inline Account::
Account( const Account &rhs )
{
   _name = rhs._name;
   _balance = rhs._balance;
   _acct_nmbr = get_unique_acct_nmbr();
}
```

Теперь почленная инициализация по умолчанию для класса Account корректно обрабатывает выделение и освобождение памяти для _name, но все еще неверно копирует номер счета, поэтому приходится кодировать явный копирующий конструктор. Однако приведенный ниже фрагмент не совсем правилен. Можете ли вы сказать, почему?

Эта реализация ошибочна, поскольку в ней не различаются инициализация и присваивание. В результате вместо вызова копирующего конструктора string мы вызываем конструктор string по умолчанию на фазе неявной инициализации и копирующий оператор присваивания string – в теле конструктора. Исправить это несложно:

```
inline Account::
Account( const Account &rhs )
       : _name( rhs._name )
{
   _balance = rhs._balance;
   _acct_nmbr = get_unique_acct_nmbr();
}
```

Самое главное – понять, что такое исправление необходимо. (Обе реализации приводят к тому, что в `_name` копируется значение из `rhs._name`, но в первой одна и та же работа выполняется дважды.) Общее эвристическое правило состоит в том, чтобы по возможности инициализировать все члены-объекты классов в списке инициализации членов.

Упражнение 14.13

Для какого определения класса скорее всего понадобится копирующий конструктор?

1. Представление `Point3w`, содержащее четыре числа с плавающей точкой.
2. Класс `matrix`, в котором память для хранения матрицы выделяется динамически в конструкторе и освобождается в деструкторе.
3. Класс `payroll` (платежная ведомость), где каждому объекту приписывается уникальный идентификатор.
4. Класс `word` (слово), содержащий объект класса `string` и вектор, в котором хранятся пары (номер строки, смещение в строке).

Упражнение 14.14

Реализуйте для каждого из данных классов копирующий конструктор, конструктор по

```
(a) class BinStrTreeNode {
   public:
      // ...
   private:
      string _value;
      int    _count;
      BinStrTreeNode *_leftchild;
      BinStrTreeNode *_rightchild;
```

умолчанию и деструктор.

```
(b) class BinStrTree {
    public:
       // ...
    private:
       BinStrTreeNode *_root;
    };

    };
```

```
(c) class iMatrix {
    public:
       // ...
    private:
       int  _rows;
       int  _cols;
       int *_matrix;
```

```
(d) class theBigMix {
    public:
       // ...
    private:
       BinStrTree    _bst;
       iMatrix       _im;
       string        _name;
       vectorMfloat> *_pvec;

    };

    };
```

Упражнение 14.15

Нужен ли копирующий конструктор для того класса, который вы выбрали в упражнении 14.3 из раздела 14.2? Если нет, объясните почему. Если да, реализуйте его.

Упражнение 14.16

Идентифицируйте в следующем фрагменте программы все места, где происходит

```
Point global;

Point foo_bar( Point arg )
{
   Point local = arg;
   Point *heap = new Point( global );
   *heap = local;
   Point pa[ 4 ] = { local, *heap };
   return *heap;
}
```

почленная инициализация:

14.7. Почленное присваивание A

Присваивание одному объекту класса значения другого объекта того же класса реализуется почленным присваиванием по умолчанию. От почленной инициализации по умолчанию оно отличается только использованием копирующего оператора присваивания вместо копирующего конструктора:

```
newAcct = oldAcct;
```

по умолчанию присваивает каждому нестатическому члену newAcct значение соответственного члена oldAcct. Компилятор генерирует следующий копирующий

```
inline Account&
Account::
operator=( const Account &rhs )
{
   _name = rhs._name;
   _balance = rhs._balance;
   _acct_nmbr = rhs._acct_nmbr;
```

оператор присваивания:

```
}
```

Как правило, если для класса не подходит почленная инициализация по умолчанию, то не подходит и почленное присваивание по умолчанию. Например, для первоначального определения класса Account, где член _name был объявлен как char*, такое присваивание не годится ни для _name, ни для _acct_nmbr.

Мы можем подавить его, если предоставим явный копирующий оператор присваивания,

```
// общий вид копирующего оператора присваивания
className&
className::
operator=( const className &rhs )
{
   // не надо присваивать самому себе
   if ( this != &rhs )
   {
        // здесь реализуется семантика копирования класса
   }
   // вернуть объект, которому присвоено значение
   return *this;
```

где будет реализована подходящая для класса семантика:

```
}
```

Здесь условная инструкция

```
if ( this != &rhs )
```

предотвращает присваивание объекта класса самому себе, что особенно неприятно в ситуации, когда копирующий оператор присваивания сначала освобождает некоторый ресурс, ассоциированный с объектом в левой части, чтобы назначить вместо него ресурс, ассоциированный с объектом в правой части. Рассмотрим копирующий оператор присваивания для класса Account:

```
Account&
Account::
operator=( const Account &rhs )
{
   // не надо присваивать самому себе
   if ( this != &rhs )
   {
      delete [] _name;
      _name = new char[strlen(rhs._name)+1];
      strcpy( _name,rhs._name );
      _balance = rhs._balance;
      _acct_nmbr = rhs._acct_nmbr;
   }
   return *this;
}
```

Когда один объект класса присваивается другому, как, например, в инструкции:

```
newAcct = oldAcct;
```

выполняются следующие шаги:

1. Выясняется, есть ли в классе явный копирующий оператор присваивания.
2. Если есть, проверяются права доступа к нему, чтобы понять, можно ли его вызывать в данном месте программы.
3. Оператор вызывается для выполнения присваивания; если же он недоступен, компилятор выдает сообщение об ошибке.
4. Если явного оператора нет, выполняется почленное присваивание по умолчанию.
5. При почленном присваивании каждому члену встроенного или составного члена объекта в левой части присваивается значение соответственного члена объекта в правой части.
6. Для каждого члена, являющегося объектом класса, рекурсивно применяются шаги 1-6, пока не останутся только члены встроенных и составных типов.

Если мы снова модифицируем определение класса Account так, что _name будет иметь тип string, то почленное присваивание по умолчанию

```
newAcct = oldAcct;
```

будет выполняться так же, как при создании компилятором следующего оператора

```
inline Account&
Account::
operator=( const Account &rhs )
{
   _balance = rhs._balance;
   _acct_nmbr = rhs._acct_nmbr;

   // этот вызов правилен и с точки зрения программиста
   name.string::operator=( rhs._name );
}
```

присваивания:

Однако почленное присваивание по умолчанию для объектов класса Account не подходит из-за _acct_nmbr. Нужно реализовать явный копирующий оператор

```
Account&
Account::
operator=( const Account &rhs )
{
   // не надо присваивать самому себе
   if ( this != &rhs )
   {
      // вызывается string::operator=( const string& )
      _name = rhs._name;
      _balance = rhs._balance;
   }
   return *this;
```

присваивания с учетом того, что _name – это объект класса string:

```
}
```

Чтобы запретить почленное копирование, мы поступаем так же, как и в случае почленной инициализации: объявляем оператор закрытым и не предоставляем его определения.

Копирующий конструктор и копирующий оператор присваивания обычно рассматривают вместе. Если необходим один, то, как правило, необходим и другой. Если запрещается один, то, вероятно, следует запретить и другой.

Упражнение 14.17

Реализуйте копирующий оператор присваивания для каждого из классов, определенных в упражнении 14.14 из раздела 14.6.

Упражнение 14.18

Нужен ли копирующий оператор присваивания для того класса, который вы выбрали в упражнении 14.3 из раздела 14.2? Если да, реализуйте его. В противном случае объясните, почему он не нужен.

14.8. Соображения эффективности A

В общем случае объект класса эффективнее передавать функции по указателю или по ссылке, нежели по значению. Например, если дана функция с сигнатурой:

```
bool sufficient_funds( Account acct, double );
```

то при каждом ее вызове требуется выполнить почленную инициализацию формального параметра acct значением фактического аргумента-объекта класса Account. Если же

```
bool sufficient_funds( Account *pacct, double );
```

функция имеет любую из таких сигнатур:

```
bool sufficient_funds( Account &acct, double );
```

то достаточно скопировать адрес объекта Account. В этом случае никакой инициализации класса не происходит (см. обсуждение взаимосвязи между ссылочными и указательными параметрами в разделе 7.3).

Хотя возвращать указатель или ссылку на объект класса также более эффективно, чем сам объект, но корректно запрограммировать это достаточно сложно. Рассмотрим такой

```
// задача решается, но для больших матриц эффективность может
// оказаться неприемлемо низкой
Matrix
operator+( const Matrix& m1, const Matrix& m2 )
{
   Matrix result;
   // выполнить арифметические операции ...
   return result;
}
```

оператор сложения:

```
Matrix a, b;
// ...

// в обоих случаях вызывается operator+()
Matrix c = a + b;
```

Этот перегруженный оператор позволяет пользователю писать

```
a = b + c;
```

Однако возврат результата по значению может потребовать слишком больших затрат времени и памяти, если Matrix представляет собой большой и сложный класс. Если эта операция выполняется часто, то она, вероятно, резко снизит производительность.

```
// более эффективно, но после возврата адрес оказывается недействительным
// это может привести к краху программы
Matrix&
operator+( const Matrix& m1, const Matrix& m2 )
{
   Matrix result;
   // выполнить сложение ...
   return result;
}
```

Следующая пересмотренная реализация намного увеличивает скорость:

но при этом происходят частые сбои программы. Дело в том, что значение переменной result не определено после выхода из функции, в которой она объявлена. (Мы возвращаем ссылку на локальный объект, который после возврата не существует.)

Значение возвращаемого адреса должно оставаться действительным после выхода из функции. В приведенной реализации возвращаемый адрес не затирается:

```
// нет возможности гарантировать отсутствие утечки памяти
// поскольку матрица может быть большой, утечки будут весьма заметными
Matrix&
operator+( const Matrix& m1, const Matrix& m2 )
{
   Matrix *result = new Matrix;
   // выполнить сложение ...
   return *result;
}
```

Однако это неприемлемо: происходит большая утечка памяти, так как ни одна из частей программы не отвечает за применение оператора `delete` к объекту по окончании его использования.

Вместо оператора сложения лучше применять именованную функцию, которой в качестве

```
// это обеспечивает нужную эффективность,
// но не является интуитивно понятным для пользователя
void
mat_add( Matrix &result,
         const Matrix& m1, const Matrix& m3 )
{
   // вычислить результат
}
```

третьего параметра передается ссылка, где следует сохранить результат:

Таким образом, проблема производительности решается, но для класса уже нельзя использовать операторный синтаксис, так что теряется возможность инициализировать

```
// более не поддерживается
```

объекты

```
Matrix c = a + b;
```

```
// тоже не поддерживается
```

и использовать их в выражениях:

```
if ( a + b > c ) ...
```

Неэффективный возврат объекта класса – слабое место C++. В качестве одного из решений предлагалось расширить язык, введя имя возвращаемого функцией объекта:

```
Matrix&
operator+( const Matrix& m1, const Matrix& m2 )
name result
{
   Matrix result;
   // ...
   return result;
}
```

Тогда компилятор мог бы самостоятельно переписать функцию, добавив к ней третий

```
// переписанная компилятором функция
// в случае принятия предлагавшегося расширения языка
void
operator+( Matrix &result, const Matrix& m1, const Matrix& m2 )
name result
{
   // вычислить результат
}
```

параметр-ссылку:

и преобразовать все вызовы этой функции, разместив результат непосредственно в области, на которую ссылается первый параметр. Например:

```
Matrix c = a + b;
```

```
Matrix c;
```

было бы трансформировано в

```
operator+(c, a, b);
```

Это расширение так и не стало частью языка, но предложенная оптимизация прижилась. Компилятор в состоянии распознать, что возвращается объект класса и выполнить трансформацию его значения и без явного расширения языка. Если дана функция общего

```
classType
functionName( paramList )
{
   classType namedResult;
   // выполнить какие-то действия ...
   return namedResult;
}
```

вида:

то компилятор самостоятельно трансформирует как саму функцию, так и все обращения к ней:

```
void
functionName( classType &namedResult, paramList )
{
   // вычислить результат и разместить его по адресу namedResult
}
```

что позволяет уйти от необходимости возвращать значение объекта и вызывать копирующий конструктор. Чтобы такая оптимизация была применена, в каждой точке возврата из функции должен возвращаться один и тот же именованный объект класса.

И последнее замечание об эффективности работы с объектами в C++. Инициализация объекта класса вида

```
Matrix c = a + b;
```

всегда эффективнее присваивания. Например, результат следующих двух инструкций

```
Matrix c;
```

такой же, как и в предыдущем случае:

```
c = a + b;
```

```
for ( int ix = 0; ix < size-2; ++ix ) {
     Matrix matSum = mat[ix] + mat[ix+1];
     // ...
```

но объем требуемых вычислений значительно больше. Аналогично эффективнее писать:

```
}
```

```
Matrix matSum;
for ( int ix = 0; ix < size-2; ++ix ) {
     matSum = mat[ix] + mat[ix+1];
     // ...
```

чем

```
}
```

Причина, по которой присваивание всегда менее эффективно, состоит в том, что возвращенный локальный объект нельзя подставить вместо объекта в левой части оператора присваивания. Иными словами, в то время как инструкцию

```
Point3d p3 = operator+( p1, p2 );
```

можно безопасно трансформировать:

```
// Псевдокод на C++
Point3d p3;
operator+( p3, p1, p2 );
```

```
Point3d p3;
```

преобразование

```
p3 = operator+( p1, p2 );
```

```
// Псевдокод на C++
// небезопасно в случае присваивания
```

в

```
operator+( p3, p1, p2 );
```

небезопасно.

Преобразованная функция требует, чтобы переданный ей объект представлял собой неформатированную область памяти. Почему? Потому что к объекту сразу применяется конструктор, который уже был применен к именованному локальному объекту. Если переданный объект уже был сконструирован, то делать это еще раз с семантической точки зрения неверно.

Что касается инициализируемого объекта, то отведенная под него память еще не подвергалась обработке. Если же объекту присваивается значение и в классе объявлены конструкторы (а именно этот случай мы и рассматриваем), можно утверждать, что эта память уже форматировалась одним из них, так что непосредственно передавать объект функции небезопасно.

Вместо этого компилятор должен создать неформатированную область памяти в виде временного объекта класса, передать его функции, а затем почленно присвоить возвращенный временный объект объекту, стоящему в левой части оператора присваивания. Наконец, если у класса есть деструктор, то он применяется к временному

```
Point3d p3;
```

объекту. Например, следующий фрагмент

```
p3 = operator+( p1, p2 );
```

```
// Псевдокод на C++
Point3d temp;
operator+( temp, p1, p2 );
p3.Point3d::operator=( temp );
```

трансформируется в такой:

```
temp.Point3d::~Point3d();
```

Майкл Тиманн (Michael Tiemann), автор компилятора GNU C++, предложил назвать это расширение языка *именованным возвращаемым значением* (return value language extension). Его точка зрения изложена в работе [LIPPMAN96b]. В нашей книге “Inside the C++ Object Model” ([LIPPMAN96a]) приводится детальное обсуждение затронутых в этой главе тем.

15

15. Перегруженные операторы и определенные пользователем преобразования

В главе 15 мы рассмотрим два вида специальных функций: перегруженные операторы и определенные пользователем преобразования. Они дают возможность употреблять объекты классов в выражениях так же интуитивно, как и объекты встроенных типов. В этой главе мы сначала изложим общие концепции проектирования перегруженных операторов. Затем представим понятие друзей класса со специальными правами доступа и обсудим, зачем они применяются, обратив особое внимание на то, как реализуются некоторые перегруженные операторы: присваивание, взятие индекса, вызов, стрелка для доступа к члену класса, инкремент и декремент, а также специализированные для класса операторы `new` и `delete`. Другая категория специальных функций, которая рассматривается в этой главе, – это функции преобразования членов (конвертеры), составляющие набор стандартных преобразований для типа класса. Они неявно применяются компилятором, когда объекты классов используются в качестве фактических аргументов функции или операндов встроенных или перегруженных операторов. Завершается глава развернутым изложением правил разрешения перегрузки функций с учетом передачи объектов в качестве аргументов, функций-членов класса и перегруженных операторов.

15.1. Перегрузка операторов

В предыдущих главах мы уже показывали, что перегрузка операторов позволяет программисту вводить собственные версии предопределенных операторов (см. главу 4) для операндов типа классов. Например, в классе `String` из раздела 3.15 задано много перегруженных операторов. Ниже приведено его определение:

```
#include <iostream>

class String;
istream& operator>>( istream &, const String & );
ostream& operator<<( ostream &, const String & );

class String {
public:
     // набор перегруженных конструкторов
     // для автоматической инициализации
          String( const char* = 0 );
          String( const String & );

     // деструктор: автоматическое уничтожение
          ~String();

     // набор перегруженных операторов присваивания
          String& operator=( const String & );
          String& operator=( const char * );

     // перегруженный оператор взятия индекса
          char& operator[]( int );

     // набор перегруженных операторов равенства
     // str1 == str2;
          bool operator==( const char * );
          bool operator==( const String & );

     // функции доступа к членам
          int size() { return _size; };
          char * c_str() { return _string; }
private:
          int   _size;
          char *_string;
};
```

В классе `String` есть три набора перегруженных операторов. Первый – это набор

```
// набор перегруженных операторов присваивания
String& operator=( const String & );
```

операторов присваивания:

```
String& operator=( const char * );
```

Сначала идет копирующий оператор присваивания. (Подробно они обсуждались в разделе 14.7.) Следующий оператор поддерживает присваивание С-строки символов

```
String name;
```

объекту типа `String`:

```
name = "Sherlock";   // использование оператора operator=( char * )
```

(Операторы присваивания, отличные от копирующих, мы рассмотрим в разделе 15.3.)

Во втором наборе есть всего один оператор – взятия индекса:

```
// перегруженный оператор взятия индекса
char& operator[]( int );
```

Он позволяет программе индексировать объекты класса String точно так же, как

```
if ( name[0] != 'S' )
```

массивы объектов встроенного типа:

```
    cout << "увы, что-то не так\n";
```

(Детально этот оператор описывается в разделе 15.4.)

В третьем наборе определены перегруженные операторы равенства для объектов класса String. Программа может проверить равенство двух таких объектов или объекта и С-

```
// набор перегруженных операторов равенства
// str1 == str2;
bool operator==( const char * );
```

строки:

```
bool operator==( const String & );
```

Перегруженные операторы позволяют использовать объекты типа класса с операторами, определенными в главе 4, и манипулировать ими так же интуитивно, как объектами встроенных типов. Например, желая определить операцию конкатенации двух объектов класса String, мы могли бы реализовать ее в виде функции-члена concat(). Но почему concat(), а не, скажем, append()? Выбранное нами имя логично и легко запоминается, но пользователь все же может забыть, как мы назвали функцию. Зачастую имя проще запомнить, если определить перегруженный оператор. К примеру, вместо concat() мы назвали бы новую операцию operator+=(). Такой оператор используется следующим

```
#include "String.h"
int main() {
   String name1 "Sherlock";
   String name2 "Holmes";

   name1 += " ";
   name1 += name2;

   if (! ( name1 == "Sherlock Holmes" ) )
      cout << "конкатенация не сработала\n";
```

образом:

```
}
```

Перегруженный оператор объявляется в теле класса точно так же, как обычная функция-член, только его имя состоит из ключевого слова operator, за которым следует один из множества предопределенных в языке C++ операторов (см. табл. 15.1). Так можно объявить operator+=( ) в классе String:

```
class String {
public:
   // набор перегруженных операторов +=
   String& operator+=( const String & );
   String& operator+=( const char * );
   // ...
private:
   // ...
};
```

```
#include <cstring>

inline String& String::operator+=( const String &rhs )
{
     // Если строка, на которую ссылается rhs, непуста
          if ( rhs._string )
          {
        String tmp( *this );

        // выделить область памяти, достаточную
        // для хранения конкатенированных строк
        _size += rhs._size;
        delete [] _string;
        _string = new char[ _size + 1 ];

        // сначала скопировать в выделенную область исходную строку
        // затем дописать в конец строку, на которую ссылается rhs
        strcpy( _string, tmp._string );
        strcpy( _string + tmp._size, rhs._string );
     }
          return *this;
}

inline String& String::operator+=( const char *s )
{
     // Если указатель s ненулевой
          if ( s )
          {
        String tmp( *this );

        // выделить область памяти, достаточную
        // для хранения конкатенированных строк
        _size += strlen( s );
        delete [] _string;
        _string = new char[ _size + 1 ];
        // сначала скопировать в выделенную область исходную строку
        // затем дописать в конец C-строку, на которую ссылается s
        strcpy( _string, tmp._string );
        strcpy( _string + tmp._size, s );
     }
          return *this;
```

и определить его следующим образом:

```
}
```

15.1.1. Члены и не члены класса

Рассмотрим операторы равенства в нашем классе String более внимательно. Первый оператор позволяет устанавливать равенство двух объектов, а второй – объекта и С-

```
#include "String.h"

int main() {
   String flower;
   // что-нибудь записать в переменную flower

   if ( flower == "lily" )  // правильно
      // ...
   else
   if ( "tulip" == flower )  // ошибка
      // ...
```

строки:

```
}
```

При первом использовании оператора равенства в main() вызывается перегруженный operator==(const char *) класса String. Однако на второй инструкции if компилятор выдает сообщение об ошибке. В чем дело?

Перегруженный оператор, являющийся членом некоторого класса, применяется только тогда, когда *левым* операндом служит объект этого класса. Поскольку во втором случае левый операнд не принадлежит к классу String, компилятор пытается найти такой встроенный оператор, для которого левым операндом может быть С-строка, а правым – объект класса String. Разумеется, его не существует, поэтому компилятор говорит об ошибке.

Но можно же создать объект класса String из С-строки с помощью конструктора класса. Почему компилятор не выполнит неявно такое преобразование:

```
if ( String( "tulip" ) == flower )  //правильно: вызывается оператор-член
```

Причина в его неэффективности. Перегруженные операторы не требуют, чтобы оба операнда имели один и тот же тип. К примеру, в классе Text определяются следующие

```
class Text {
public:
   Text( const char * = 0 );
   Text( const Text & );

   // набор перегруженных операторов равенства
   bool operator==( const char * ) const;
   bool operator==( const String & ) const;
   bool operator==( const Text & ) const;

   // ...
```

операторы равенства:

```
};
```

и выражение в main() можно переписать так:

```
if ( Text( "tulip" ) == flower )  // вызывается Text::operator==()
```

Следовательно, чтобы найти подходящий для сравнения оператор равенства, компилятору придется просмотреть все определения классов в поисках конструктора, способного привести левый операнд к некоторому типу класса. Затем для каждого из таких типов нужно проверить все ассоциированные с ним перегруженные операторы равенства, чтобы понять, может ли хоть один из них выполнить сравнение. А после этого компилятор должен решить, какая из найденных комбинаций конструктора и оператора равенства (если таковые нашлись) лучше всего соответствует операнду в правой части! Если потребовать от компилятора выполнения всех этих действий, то время трансляции программ C++ резко возрастет. Вместо этого компилятор просматривает только перегруженные операторы, определенные как члены класса левого операнда (и его базовых классов, как мы покажем в главе 19).

Разрешается, однако, определять перегруженные операторы, не являющиеся членами класса. При анализе строки в `main()`, вызвавшей ошибку компиляции, подобные операторы принимались во внимание. Таким образом, сравнение, в котором C-строка стоит в левой части, можно сделать корректным, если заменить операторы равенства, являющиеся членами класса `String`, на операторы равенства, объявленные в области

```
bool operator==( const String &, const String & );
```

видимости пространства имен:

```
bool operator==( const String &, const char * );
```

Обратите внимание, что эти глобальные перегруженные операторы имеют на один параметр больше, чем операторы-члены. Если оператор является членом класса, то первым параметром неявно передается указатель `this`. То есть для операторов-членов выражение

```
flower == "lily"
```

переписывается компилятором в виде:

```
flower.operator==( "lily" )
```

и на левый операнд `flower` в определении перегруженного оператора-члена можно сослаться с помощью `this`. (Указатель `this` введен в разделе 13.4.) В случае глобального перегруженного оператора параметр, представляющий левый операнд, должен быть задан явно.

Тогда выражение

```
flower == "lily"
```

вызывает оператор

```
bool operator==( const String &, const char * );
```

Непонятно, какой оператор вызывается для второго случая использования оператора равенства:

```
"tulip" == flower
```

Мы ведь не определили такой перегруженный оператор:

```
bool operator==( const char *, const String & );
```

Но это необязательно. Когда перегруженный оператор является функцией в пространстве имен, то как для первого, так и для второго его параметра (для левого и правого операндов) рассматриваются возможные преобразования, т.е. компилятор интерпретирует второе использование оператора равенства как

```
operator==( String("tulip"), flower );
```

и вызывает для выполнения сравнения следующий перегруженный оператор:

```
bool operator==( const String &, const String & );
```

Но тогда зачем мы предоставили второй перегруженный оператор:

```
bool operator==( const String &, const char * );
```

Преобразование типа из С-строки в класс `String` может быть применено и к правому операнду. Функция `main()` будет компилироваться без ошибок, если просто определить в пространстве имен перегруженный оператор, принимающий два операнда `String`:

```
bool operator==( const String &, const String & );


bool operator==( const char *, const String & );
```

Предоставлять ли только этот оператор или еще два:

```
bool operator==( const String &, const char * );
```

зависит от того, насколько велики затраты на преобразование из С-строки в `String` во время выполнения, то есть от “стоимости” дополнительных вызовов конструктора в программах, пользующихся нашим классом `String`. Если оператор равенства будет часто использоваться для сравнения С-строк и объектов , то лучше предоставить все три варианта. (Мы вернемся к вопросу эффективности в разделе, посвященном друзьям.

Подробнее о приведении к типу класса с помощью конструкторов мы расскажем в разделе 15.9; в разделе 15.10 речь пойдет о разрешении перегрузки функций с помощью описанных преобразований, а в разделе 15.12 – о разрешении перегрузки операторов.)

Итак, на основе чего принимается решение, делать ли оператор членом класса или членом пространства имен? В некоторых случаях у программиста просто нет выбора:

- если перегруженный оператор является членом класса, то он вызывается лишь при условии, что левым операндом служит член этого класса. Если же левый операнд имеет другой тип, оператор *обязан* быть членом пространства имен;
- язык требует, чтобы операторы присваивания (`"="`), взятия индекса (`"[]"`), вызова (`"()"`) и доступа к членам по стрелке (`"->"`) были определены как члены класса. В противном случае выдается сообщение об ошибке компиляции:

```
// ошибка: должен быть членом класса
char& operator[]( String &, int ix );
```

(Подробнее оператор присваивания рассматривается в разделе 15.3, взятия индекса – в разделе 15.4, вызова – в разделе 15.5, а оператор доступа к члену по стрелке – в разделе 15.6.)

В остальных случаях решение принимает проектировщик класса. Симметричные операторы, например оператор равенства, лучше определять в пространстве имен, если членом класса может быть любой операнд (как в `String`).

Прежде чем закончить этот подраздел, определим операторы равенства для класса

```
bool operator==( const String &str1, const String &str2 )
{
   if ( str1.size() != str2.size() )
      return false;
   return strcmp( str1.c_str(), str2.c_str() ) ? false : true ;
}

inline bool operator==( const String &str, const char *s )
{
   return strcmp( str.c_str(), s ) ? false : true ;
```

`String` в пространстве имен:

```
}
```

15.1.2. Имена перегруженных операторов

Перегружать можно только предопределенные операторы языка C++ (см. табл. 15.1).

Таблица 15.1. Перегружаемые операторы

+	-	*	/	%	^	&	|	~
!	,	=	<	>	<=	>=	++	--
<<	>>	==	!=	&&	||	+=	-=	/=
%=	^=	&=	|=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]			

Проектировщик класса не вправе объявить перегруженным оператор с другим именем. Так, при попытке объявить оператор ** для возведения в степень компилятор выдаст сообщение об ошибке.

```
// неперегружаемые операторы
```

Следующие четыре оператора языка C++ не могут быть перегружены:

```
:: .* . ?:
```

Предопределенное назначение оператора нельзя изменить для встроенных типов. Например, не разрешается переопределить встроенный оператор сложения целых чисел

```
// ошибка: нельзя переопределить встроенный оператор сложения int
```

так, чтобы он проверял результат на переполнение.

```
int operator+( int, int );
```

Нельзя также определять дополнительные операторы для встроенных типов данных, например добавить к множеству встроенных операций `operator+` для сложения двух массивов.

Перегруженный оператор определяется исключительно для операндов типа класса или перечисления и может быть объявлен только как член класса или пространства имен, принимая хотя бы один параметр типа класса или перечисления (переданный по значению или по ссылке).

Предопределенные приоритеты операторов (см. раздел 4.13) изменить нельзя. Независимо от типа класса и реализации оператора в инструкции

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

всегда сначала выполняется `operator+`, а затем `operator==`; однако помощью скобок порядок можно изменить.

Предопределенная арность операторов также должна быть сохранена. К примеру, унарный логический оператор НЕ нельзя определить как бинарный оператор для двух объектов класса `String`. Следующая реализация некорректна и приведет к ошибке

```
// некорректно: ! - это унарный оператор
bool operator!( const String &s1, const String &s2 )
{
   return ( strcmp( s1.c_str(), s2.c_str() ) != 0 );
```

компиляции:

```
}
```

Для встроенных типов четыре предопределенных оператора (`"+"`, `"-"`, `"*"` и `"&"`) используются либо как унарные, либо как бинарные. В любом из этих качеств они могут быть перегружены.

Для всех перегруженных операторов, за исключением `operator()`, недопустимы аргументы по умолчанию.

15.1.3. Разработка перегруженных операторов

Операторы присваивания, взятия адреса и оператор “запятая” имеют предопределенный смысл, если операндами являются объекты типа класса. Но их можно и перегружать. Семантика всех остальных операторов, когда они применяются к таким операндам, должна быть явно задана разработчиком. Выбор предоставляемых операторов зависит от ожидаемого использования класса.

Начинать следует с определения его открытого интерфейса. Набор открытых функций-членов формируется с учетом операций, которые класс должен предоставлять пользователям. Затем принимается решение, какие функции стоит реализовать в виде перегруженных операторов.

После определения открытого интерфейса класса проверьте, есть ли логическое соответствие между операциями и операторами:

- `isEmpty()` становится оператором "ЛОГИЧЕСКОЕ НЕ", `operator!()`.
- `isEqual()` становится оператором равенства, `operator==()`.
- `copy()` становится оператором присваивания, `operator=()`.

У каждого оператора есть некоторая естественная семантика. Так, бинарный `+` всегда ассоциируется со сложением, а его отображение на аналогичную операцию с классом может оказаться удобной и краткой нотацией. Например, для матричного типа сложение двух матриц является вполне подходящим расширением бинарного плюса.

Примером неправильного использования перегрузки операторов является определение `operator+()` как операции вычитания, что бессмысленно: не согласующаяся с интуицией семантика опасна.

Такой оператор одинаково хорошо поддерживает несколько различных интерпретаций. Безупречно четкое и обоснованное объяснение того, что делает `operator+()`, вряд ли устроит пользователей класса `String`, полагающих, что он служит для конкатенации строк. Если семантика перегруженного оператора неочевидна, то лучше его не предоставлять.

Эквивалентность семантики составного оператора и соответствующей последовательности простых операторов для встроенных типов (например, эквивалентность оператора `+`, за которым следует `=`, и составного оператора `+=`) должна быть явно поддержана и для класса. Предположим, для `String` определены как `operator+()`, так и `operator=()` для поддержки операций конкатенации и почленного

```
String s1( "C" );
String s2( "++" );
```

копирования:

```
s1 = s1 + s2;    // s1 == "C++"
```

Но этого *недостаточно* для поддержки составного оператора присваивания

```
s1 += s2;
```

Его следует определить явно, так, чтобы он поддерживал ожидаемую семантику.

Упражнение 15.1

Почему при выполнении следующего сравнения *не* вызывается перегруженный оператор `operator==(const String&, const String&)`:

```
"cobble" == "stone"
```

Упражнение 15.2

Напишите перегруженные операторы неравенства, которые могут быть использованы в

```
String != String
String != C-строка
```

таких сравнениях:

```
C-строка != String
```

Объясните, почему вы решили реализовать один или несколько операторов.

Упражнение 15.3

Выявите те функции-члены класса Screen, реализованного в главе 13 (разделы 13.3, 13.4 и 13.6), которые можно перегружать.

Упражнение 15.4

Объясните, почему перегруженные операторы ввода и вывода, определенные для класса String из раздела 3.15, объявлены как глобальные функции, а не функции-члены.

Упражнение 15.5

Реализуйте перегруженные операторы ввода и вывода для класса Screen из главы 13.

15.2. Друзья

Рассмотрим еще раз перегруженные операторы равенства для класса String, определенные в области видимости пространства имен. Оператор равенства для двух

```
bool operator==( const String &str1, const String &str2 )
{
   if ( str1.size() != str2.size() )
      return false;
   return strcmp( str1.c_str(), str2.c_str() ) ? false : true;
```

объектов String выглядит следующим образом:

```
}
```

```
bool String::operator==( const String &rhs ) const
{
   if ( _size != rhs._size )
      return false;
   return strcmp( _string, rhs._string ) ? false : true;
```

Сравните это определение с определением того же оператора как функции-члена:

```
}
```

Нам пришлось модифицировать способ обращения к закрытым членам класса String. Поскольку новый оператор равенства – это глобальная функция, а не функция-член, у него нет доступа к закрытым членам класса String. Для получения размера объекта String и лежащей в его основе C-строки символов используются функции-члены size() и c_str().

Альтернативной реализацией является объявление глобальных операторов равенства *друзьями* класса String. Если функция или оператор объявлены таким образом, им предоставляется доступ к неоткрытым членам.

Объявление друга (оно начинается с ключевого слова friend) встречается только внутри определения класса. Поскольку друзья не являются членами класса, объявляющего дружественные отношения, то безразлично, в какой из секций – public, private или protected – они объявлены. В примере ниже мы решили поместить все подобные

```
class String {
   friend bool operator==( const String &, const String & );
   friend bool operator==( const char *, const String & );
   friend bool operator==( const String &, const char * );
public:
   // ... остальная часть класса String
```

объявления сразу после заголовка класса:

```
};
```

В этих трех строчках три перегруженных оператора сравнения, принадлежащие глобальной области видимости, объявляются друзьями класса String, а следовательно, в

```
// дружественные операторы напрямую обращаются к закрытым членам
// класса String
bool operator==( const String &str1, const String &str2 )
{
   if ( str1._size != str2._size )
      return false;
   return strcmp( str1._string, str2._string ) ? false : true;
```

их определениях можно напрямую обращаться к закрытым членам данного класса:

```
inline bool operator==( const String &str, const char *s )
{
   return strcmp( str._string, s ) ? false : true;
}
```

```
}
// и т.д.
```

Можно возразить, что в данном случае прямой доступ к членам _size и _string необязателен, так как встроенные функции c_str() и size() столь же эффективны и при этом сохраняют инкапсуляцию, а значит, нет особой нужды объявлять операторы равенства для класса String его друзьями.

Как узнать, следует ли сделать оператор, не являющийся членом класса, его другом или воспользоваться функциями доступа? В общем случае разработчик должен сократить до минимума число объявленных функций и операторов, которые имеют доступ к внутреннему представлению класса. Если имеются функции доступа, обеспечивающие равную эффективность, то предпочтение следует отдать им, тем самым изолируя операторы в пространстве имен от изменений представления класса, как это делается и для других функций. Если же разработчик класса не предоставляет функций доступа для некоторых членов, а объявленный в пространстве имен оператор должен к этим членам обращаться, то использование механизма друзей становится неизбежным.

Наиболее часто такой механизм применяется для того, чтобы разрешить перегруженным операторам, не являющимся членами класса, доступ к его закрытым членам. Если бы не необходимость обеспечить симметрию левого и правого операндов, то перегруженный оператор был бы функцией-членом с полными правами доступа.

Хотя объявления друзей обычно употребляются по отношению к операторам, бывают случаи, когда функцию в пространстве имен, функцию-член другого класса или даже целый класс приходится объявлять таким образом. Если один класс объявлен другом второго, то все функции-члены первого класса получают доступ к неоткрытым членам другого. Рассмотрим это на примере функций, не являющихся операторами.

Класс должен объявлять другом каждую из множества перегруженных функций, которой

```
extern ostream& storeOn( ostream &, Screen & );
extern BitMap& storeOn( BitMap &, Screen & );
// ...

class Screen
{
   friend ostream& storeOn( ostream &, Screen & );
   friend BitMap& storeOn( BitMap &, Screen & );
   // ...
```

он хочет дать неограниченные права доступа:

```
};
```

Если функция манипулирует объектами двух разных классов и ей нужен доступ к их неоткрытым членам, то такую функцию можно либо объявить другом обоих классов, либо сделать членом одного и другом второго.

```
class Window;   // это всего лишь объявление
class Screen {
   friend bool is_equal( Screen &, Window & );
   // ...
};

class Window {
   friend bool is_equal( Screen &, Window & );
   // ...
```

Объявление функции другом двух классов должно выглядеть так:

```
};
```

Если же мы решили сделать функцию членом одного класса и другом второго, то объявления будут построены следующим образом:

```
class Window;
class Screen {
   // copy() - член класса Screen
   Screen& copy( Window & );
   // ...
};

class Window {
   // Screen::copy() - друг класса Window
   friend Screen& Screen::copy( Window & );
   // ...
};
```

```
Screen& Screen::copy( Window & ) { /* ... */ }
```

Функция-член одного класса не может быть объявлена другом второго, пока компилятор не увидел определения ее собственного класса. Это не всегда возможно. Предположим, что `Screen` должен объявить некоторые функции-члены `Window` своими друзьями, а `Window` – объявить таким же образом некоторые функции-члена `Screen`. В таком случае

```
class Window;
class Screen {
   friend class Window;
   // ...
```

весь класс `Window` объявляется другом `Screen`:

```
};
```

К закрытым членам класса `Screen` теперь можно обращаться из любой функции-члена `Window`.

Упражнение 15.6

Реализуйте операторы ввода и вывода, определенные для класса `Screen` в упражнении 15.5, в виде друзей и модифицируйте их определения так, чтобы они напрямую обращались к закрытым членам. Какая реализация лучше? Объясните почему.

15.3. Оператор =

Присваивание одного объекта другому объекту того же класса выполняется с помощью копирующего оператора присваивания. (Этот специальный случай был рассмотрен в разделе 14.7.)

Для класса могут быть определены и другие операторы присваивания. Если объектам класса надо присваивать значения типа, отличного от этого класса, то разрешается определить такие операторы, принимающие подобные параметры. Например, чтобы

```
String car ("Volks");
```

поддержать присваивание С-строки объекту `String`:

```
car = "Studebaker";
```

мы предоставляем оператор, принимающий параметр типа const char*. Эта операция

```
class String {
public:
   // оператор присваивания для char*
   String& operator=( const char * );
   // ...
private:
   int  _size;
   char *string;
```

уже была объявлена в нашем классе:

```
};
```

Такой оператор реализуется следующим образом. Если объекту String присваивается нулевой указатель, он становится "пустым". В противном случае ему присваивается

```
String& String::operator=( const char *sobj )
{
   // sobj - нулевой указатель
   if (! sobj ) {
      _size = 0;
      delete[] _string;
      _string = 0;
   }
   else {
      _size = strlen( sobj );
      delete[] _string;
      _string = new char[ _size + 1 ];
      strcpy( _string, sobj );
   }
   return *this;
```

копия С-строки:

```
}
```

_string ссылается на копию той С-строки, на которую указывает sobj. Почему на копию? Потому что непосредственно присвоить sobj члену _string нельзя:

```
_string = sobj;   // ошибка: несоответствие типов
```

sobj – это указатель на const и, следовательно, не может быть присвоен указателю на "не-const" (см. раздел 3.5). Изменим определение оператора присваивания:

```
String& String::operator=( const *sobj ) { // ... }
```

Теперь _string прямо ссылается на С-строку, адресованную sobj. Однако при этом возникают другие проблемы. Напомним, что С-строка имеет тип const char*. Определение параметра как указателя на не-const делает присваивание невозможным:

```
car = "Studebaker";  // недопустимо с помощью operator=( char *) !
```

Итак, выбора нет. Чтобы присвоить С-строку объекту типа String, параметр должен иметь тип const char*.

Хранение в _string прямой ссылки на С-строку, адресуемую sobj, порождает и иные сложности. Мы не знаем, на что именно указывает sobj. Это может быть массив

```
char ia[] = { 'd', 'a', 'n', 'c', 'e', 'r' };
String trap = ia;   // trap._string ссылается на ia
```

символов, который модифицируется способом, неизвестным объекту String. Например:

```
ia[3] = 'g';  // а вот это нам не нужно:
              // модифицируется и ia, и trap._string
```

Если trap._string напрямую ссылался на ia, то объект trap демонстрировал бы своеобразное поведение: его значение может изменяться без вызова функций-членов класса String. Поэтому мы полагаем, что выделение области памяти для хранения копии значения С-строки менее опасно.

Обратите внимание, что в операторе присваивания используется delete. Член _string содержит ссылку на массив символов, расположенный в хипе. Чтобы предотвратить утечку, память, выделенная под старую строку, освобождается с помощью delete до выделения памяти под новую. Поскольку _string адресует массив символов, следует использовать версию delete для массивов (см. раздел 8.4).

И последнее замечание об операторе присваивания. Тип возвращаемого им значения – это ссылка на класс String. Почему именно ссылка? Дело в том, что для встроенных

```
// сцепление операторов присваивания
int iobj, jobj;
```

типов операторы присваивания можно сцеплять:

```
iobj = jobj = 63;
```

Они ассоциируются справа налево, т.е. в предыдущем примере присваивания выполняются так:

```
iobj = (jobj = 63);
```

Это удобно и при работе с объектами класса String: поддерживается, к примеру,

```
String ver, noun;
```

следующая конструкция:

```
verb = noun = "count";
```

При первом присваивании из этой цепочки вызывается определенный ранее оператор для const char*. Тип полученного результата должен быть таким, чтобы его можно было использовать как аргумент для копирующего оператора присваивания класса String. Поэтому, хотя параметр данного оператора имеет тип const char *, возвращается все же ссылка на String.

Операторы присваивания бывают перегруженными. Например, в нашем классе String есть такой набор:

```
// набор перегруженных операторов присваивания
String& operator=( const String & );

String& operator=( const char * );
```

Отдельный оператор присваивания может существовать для каждого типа, который разрешено присваивать объекту `String`. Однако все такие операторы должны быть определены как функции-члены класса.

15.4. Оператор взятия индекса

Оператор взятия индекса `operator[]()` можно определять для классов, представляющих абстракцию контейнера, из которого извлекаются отдельные элементы. Примерами таких контейнеров могут служить наш класс `String`, класс `IntArray`, представленный в главе 2, или шаблон класса `vector`, определенный в стандартной библиотеке C++. Оператор взятия индекса обязан быть функцией-членом класса.

У пользователей `String` должна иметься возможность чтения и записи отдельных символов члена `_string`. Мы хотим поддержать следующий способ применения объектов

```
String entry( "extravagant" );
String mycopy;

for ( int ix = 0; ix < entry.size(); ++ix )
```

данного класса:

```
    mycopy[ ix ] = entry[ ix ];
```

Оператор взятия индекса может появляться как слева, так и справа от оператора присваивания. Чтобы быть в левой части, он должен возвращать l-значение

```
#include <cassert>

inine char&
String::operator[]( int elem ) const
{
   assert( elem >= 0 && elem < _size );
   return _string[ elem ];
```

индексируемого элемента. Для этого мы возвращаем ссылку:

```
}


String color( "violet" );
```

В следующем фрагменте нулевому элементу массива `color` присваивается символ `'V'`:

```
color[ 0 ] = 'V';
```

Обратите внимание, что в определении оператора проверяется выход индекса за границы массива. Для этого используется библиотечная C-функция `assert()`. Можно также возбудить исключение, показывающее, что значение `elem` меньше 0 или больше длины C-

строки, на которую ссылается _string. (Возбуждение и обработка исключений обсуждались в главе 11.)

15.5. Оператор вызова функции

Оператор вызова функции может быть перегружен для объектов типа класса. (Мы уже видели, как он используется, при рассмотрении объектов-функций в разделе 12.3.) Если определен класс, представляющий некоторую операцию, то для ее вызова перегружается соответствующий оператор. Например, для взятия абсолютного значения числа типа int

```
class absInt {
public:
   int operator()( int val ) {
      int result = val < 0 ? -val : val;
      return result;
   }
```

можно определить класс absInt:

```
};
```

Перегруженный оператор operator() должен быть объявлен как функция-член с произвольным числом параметров. Параметры и возвращаемое значение могут иметь любые типы, допустимые для функций (см. разделы 7.2, 7.3 и 7.4). operator() вызывается путем применения списка аргументов к объекту того класса, в котором он определен. Мы рассмотрим, как он используется в одном из обобщенных алгоритмов, описанных в главе 12. В следующем примере обобщенный алгоритм transform() вызывается для применения определенной в absInt операции к каждому элементу

```
#include <vector>
#include <algoritm>

int main() {
   int ia[] = { -0, 1, -1, -2, 3, 5, -5, 8 };
   vector< int > ivec( ia, ia+8 );

   // заменить каждый элемент его абсолютным значением
   transform( ivec.begin(), ivec.end(), ivec.begin(), absInt() );

   // ...
```

вектора ivec, т.е. для замены элемента его абсолютным значением.

```
}
```

Первый и второй аргументы transform() ограничивают диапазон элементов, к которым применяется операция absInt. Третий указывает на начало вектора, где будет сохранен результат применения операции.

Четвертый аргумент – это временный объект класса absInt, создаваемый с помощью конструктора по умолчанию. Конкретизация обобщенного алгоритма transform(), вызываемого из main(), могла бы выглядеть так:

```
typedef vector< int >::iterator iter_type;

// конкретизация transform()
// операция absInt применяется к элементу вектора int

iter_type transform( iter_type iter, iter_type last,
                     iter_type result, absInt func )
{
   while ( iter != last )
      *result++ = func( *iter++ );  // вызывается absInt::operator()

   return iter;
}
```

func – это объект класса, который предоставляет операцию absInt, заменяющую число типа int его абсолютным значением. Он используется для вызова перегруженного оператора operator() класса absInt. Этому оператору передается аргумент *iter, указывающий на тот элемент вектора, для которого мы хотим получить абсолютное значение.

15.6. Оператор “стрелка”

Оператор “стрелка”, разрешающий доступ к членам, может перегружаться для объектов класса. Он должен быть определен как функция-член и обеспечивать семантику указателя. Чаще всего этот оператор используется в классах, которые предоставляют “интеллектуальный указатель” (smart pointer), ведущий себя аналогично встроенным, но поддерживают и некоторую дополнительную функциональность.

Допустим, мы хотим определить тип класса для представления указателя на объект

```
class ScreenPtr {
   // ...
private:
   Screen *ptr;
```

Screen (см. главу 13):

```
};
```

Определение ScreenPtr должно быть таким, чтобы объект этого класса гарантировано указывал на объект Screen: в отличие от встроенного указателя, он не может быть нулевым. Тогда приложение сможет пользоваться объектами типа ScreenPtr, не проверяя, указывают ли они на какой-нибудь объект Screen. Для этого нужно определить класс ScreenPtr с конструктором, но без конструктора по умолчанию

```
class ScreenPtr {
public:
   ScreenPtr( const Screen &s ) : ptr( &s ) { }
   // ...
```

(детально конструкторы рассматривались в разделе 14.2):

```
};
```

В любом определении объекта класса ScreenPtr должен присутствовать

```
ScreenPtr p1;   // ошибка: у класса ScreenPtr нет конструктора по
          умолчанию

Screen myScreen( 4, 4 );
```

инициализатор – объект класса Screen, на который будет ссылаться объект ScreenPtr:

```
ScreenPtr ps( myScreen );  // правильно
```

Чтобы класс ScreenPtr вел себя как встроенный указатель, необходимо определить некоторые перегруженные операторы – разыменования (*) и “стрелку” для доступа к

```
// перегруженные операторы для поддержки поведения указателя
class ScreenPtr {
public:
   Screen& operator*()  { return *ptr; }
   Screen* operator->() { return ptr; }
   // ...
```

членам:

```
};
```

Оператор доступа к членам унарный, поэтому параметры ему не передаются. При использовании в составе выражения его результат зависит только от типа левого операнда. Например, в инструкции

```
point->action();
```

исследуется тип point. Если это указатель на некоторый тип класса, то применяется семантика встроенного оператора доступа к члену. Если же это объект или ссылка на объект, то проверяется, есть ли в этом классе перегруженный оператор доступа. Когда перегруженный оператор “стрелка” определен, он вызывается для объекта point, иначе инструкция неверна, поскольку для обращения к членам самого объекта (в том числе по ссылке) следует использовать оператор “точка”.

Перегруженный оператор “стрелка” должен возвращать либо указатель на тип класса, либо объект класса, в котором он определен. Если возвращается указатель, то к нему применяется семантика встроенного оператора “стрелка”. В противном случае процесс продолжается рекурсивно, пока не будет получен указатель или определена ошибка. Например, так можно воспользоваться объектом ps класса ScreenPtr для доступа к членам Screen:

```
ps->move( 2, 3 );
```

Поскольку слева от оператора “стрелка” находится объект типа ScreenPtr, то употребляется перегруженный оператор этого класса, который возвращает указатель на объект Screen. Затем к полученному значению применяется встроенный оператор “стрелка” для вызова функции-члена move().

Ниже приводится небольшая программа для тестирования класса ScreenPtr. Объект типа ScreenPtr используется точно так же, как любой объект типа Screen*:

```
#include <iostream>
#include <string>
#include "Screen.h"

void printScreen( const ScreenPtr &ps )
{
   cout << "Screen Object ( "
        << ps->height() << ", "
        << ps->width() << " )\n\n";

   for ( int ix = 1; ix <= ps->height(); ++ix )
   {
      for ( int iy = 1; iy <= ps->width(); ++iy )
         cout << ps->get( ix, iy );
      cout << "\n";
    }
}

int main() {
   Screen sobj( 2, 5 );
   string init( "HelloWorld" );
   ScreenPtr ps( sobj );

   // Установить содержимое экрана
   string::size_type initpos = 0;
   for ( int ix = 1; ix <= ps->height(); ++ix )
      for ( int iy = 1; iy <= ps->width(); ++iy )
      {
         ps->move( ix, iy );
         ps->set( init[ initpos++ ] );
      }

    // Вывести содержимое экрана
    printScreen( ps );

    return 0;
}
```

Разумеется, подобные манипуляции с указателями на объекты классов не так эффективны, как работа со встроенными указателями. Поэтому интеллектуальный указатель должен предоставлять дополнительную функциональность, важную для приложения, чтобы оправдать сложность своего использования.

15.7. Операторы инкремента и декремента

Продолжая развивать реализацию класса `ScreenPtr`, введенного в предыдущем разделе, рассмотрим еще два оператора, которые поддерживаются для встроенных указателей и которые желательно иметь и для нашего интеллектуального указателя: инкремент (`++`) и декремент (`--`). Чтобы использовать класс `ScreenPtr` для ссылки на элементы массива объектов `Screen`, туда придется добавить несколько дополнительных членов.

Сначала мы определим новый член `size`, который содержит либо нуль (это говорит о том, что объект `ScreenPtr` указывает на единственный объект), либо размер массива, адресуемого объектом `ScreenPtr`. Нам также понадобится член `offset`, запоминающий смещение от начала данного массива:

```
class ScreenPtr {
public:
   // ...
private:
   int size;         // размер массива: 0, если единственный объект
   int offset;       // смещение ptr от начала массива
   Screen *ptr;
};
```

Модифицируем конструктор класса ScreenPtr с учетом его новой функциональности и дополнительных членов,. Пользователь нашего класса должен передать конструктору

```
class ScreenPtr {
public:
          ScreenPtr( Screen &s , int arraySize = 0 )
              : ptr( &s ), size ( arraySize ), offset( 0 ) { }
private:
   int size;
   int offset;
   Screen *ptr;
};
```

дополнительный аргумент, если создаваемый объект указывает на массив:

С помощью этого аргумента задается размер массива. Чтобы сохранить прежнюю функциональность, предусмотрим для него значение по умолчанию, равное нулю. Таким образом, если второй аргумент конструктора опущен, то член size окажется равен 0 и, следовательно, такой объект будет указывать на единственный объект Screen. Объекты

```
Screen myScreen( 4, 4 );
ScreenPtr pobj( myScreen );  // правильно: указывает на один объект
const int arrSize = 10;
Screen *parray = new Screen[ arrSize ];
```

нового класса ScreenPtr можно определять следующим образом:

```
ScreenPtr parr( *parray, arrSize );  // правильно: указывает на массив
```

Теперь мы готовы определить в ScreenPtr перегруженные операторы инкремента и декремента. Однако они бывают двух видов: префиксные и постфиксные. К счастью, можно определить оба варианта. Для префиксного оператора объявление не содержит

```
class ScreenPtr {
public:
   Screen& operator++();
   Screen& operator--();
   // ...
```

ничего неожиданного:

```
};
```

Такие операторы определяются как унарные операторные функции. Использовать префиксный оператор инкремента можно, к примеру, следующим образом:

```
const int arrSize = 10;
Screen *parray = new Screen[ arrSize ];
ScreenPtr parr( *parray, arrSize );

for ( int ix = 0;
      ix < arrSize;
      ++ix, ++parr )  // эквивалентно parr.operator++()
    }
```

```
      printScreen( parr );
```

```
Screen& ScreenPtr::operator++()
{
   if ( size == 0 ) {
      cerr << "не могу инкрементировать указатель для одного объекта\n";
      return *ptr;
   }
   if ( offset >= size - 1 ) {
      cerr << "уже в конце массива\n";
      return *ptr;
   }

   ++offset;
   return *++ptr;
}

Screen& ScreenPtr::operator--()
{
   if ( size == 0 ) {
      cerr << "не могу декрементировать указатель для одного объекта\n";
      return *ptr;
   }
   if ( offset <= 0 ) {
      cerr << "уже в начале массива\n";
      return *ptr;
   }

   --offset;
   return *--ptr;
```

Определения этих перегруженных операторов приведены ниже:

```
}
```

Чтобы отличить префиксные операторы от постфиксных, в объявлениях последних имеется дополнительный параметр типа `int`. В следующем фрагменте объявлены префиксные и постфиксные варианты операторов инкремента и декремента для класса

```
class ScreenPtr {
public:
   Screen& operator++();    // префиксные операторы
   Screen& operator--();
   Screen& operator++(int); // постфиксные операторы
   Screen& operator--(int);
   // ...
```

`ScreenPtr`:

```
};
```

```
Screen& ScreenPtr::operator++(int)
{
   if ( size == 0 ) {
      cerr << "не могу инкрементировать указатель для одного объекта\n";
      return *ptr;
   }
   if ( offset == size ) {
      cerr << "уже на один элемент дальше конца массива\n";
      return *ptr;
   }

   ++offset;
   return *ptr++;
}

Screen& ScreenPtr::operator--(int)
{
   if ( size == 0 ) {
      cerr << "не могу декрементировать указатель для одного объекта\n";
      return *ptr;
   }
   if ( offset == -1 ) {
      cerr << "уже на один элемент раньше начала массива\n";
      return *ptr;
   }

   --offset;
   return *ptr--;
}
```

Ниже приведена возможная реализация постфиксных операторов:

Обратите внимание, что давать название второму параметру нет необходимости, поскольку внутри определения оператора он не употребляется. Компилятор сам подставляет для него значение по умолчанию, которое можно игнорировать. Вот пример

```
const int arrSize = 10;
Screen *parray = new Screen[ arrSize ];
ScreenPtr parr( *parray, arrSize );

for ( int ix = 0; ix < arrSize; ++ix)
```

использования постфиксного оператора:

```
    printScreen( parr++ );
```

При его явном вызове необходимо все же передать значение второго целого аргумента. В случае нашего класса `ScreenPtr` это значение игнорируется, поэтому может быть любым:

```
parr.operator++(1024);   // вызов постфиксного operator++
```

Перегруженные операторы инкремента и декремента разрешается объявлять как дружественные функции. Изменим соответствующим образом определение класса `ScreenPtr`:

```
class ScreenPtr {
   // объявления не членов
   friend Screen& operator++( Screen & );    // префиксные операторы
   friend Screen& operator--( Screen & );
   friend Screen& operator++( Screen &, int); // постфиксные операторы
   friend Screen& operator--( Screen &, int);
public:
   // определения членов
};
```

Упражнение 15.7

Напишите определения перегруженных операторов инкремента и декремента для класса `ScreenPtr`, предположив, что они объявлены как друзья класса.

Упражнение 15.8

С помощью `ScreenPtr` можно представить указатель на массив объектов класса `Screen`. Модифицируйте перегруженные `operator*()` и `operator->()` (см. раздел 15.6) так, чтобы указатель ни при каком условии не адресовал элемент перед началом или за концом массива. Совет: в этих операторах следует воспользоваться новыми членами `size` и `offset`.

15.8. Операторы new и delete

По умолчанию выделение объекта класса из хипа и освобождение занятой им памяти выполняются с помощью глобальных операторов `new()` и `delete()`, определенных в стандартной библиотеке C++. (Мы рассматривали эти операторы в разделе 8.4.) Но класс может реализовать и собственную стратегию управления памятью, предоставив одноименные операторы-члены. Если они определены в классе, то вызываются вместо глобальных операторов с целью выделения и освобождения памяти для объектов этого класса.

Определим операторы `new()` и `delete()` в нашем классе `Screen`.

Оператор-член `new()` должен возвращать значение типа `void*` и принимать в качестве первого параметра значение типа `size_t`, где `size_t` – это `typedef`, определенный в

```
class Screen {
public:
   void *operator new( size_t );
   // ...
};
```

системном заголовочном файле `<cstddef>`. Вот его объявление:

Когда для создания объекта типа класса используется `new()`, компилятор проверяет, определен ли в этом классе такой оператор. Если да, то для выделения памяти под объект вызывается именно он, в противном случае – глобальный оператор `new()`. Например, следующая инструкция

```
Screen *ps = new Screen;
```

создает объект Screen в хипе, а поскольку в этом классе есть оператор new(), то вызывается он. Параметр size_t оператора автоматически инициализируется значением, равным размеру Screen в байтах.

Добавление оператора new() в класс или его удаление оттуда не отражаются на пользовательском коде. Вызов new выглядит одинаково как для глобального оператора, так и для оператора-члена. Если бы в классе Screen не было собственного new(), то обращение осталось бы правильным, только вместо оператора-члена вызывался бы глобальный оператор.

С помощью оператора разрешения глобальной области видимости можно вызвать глобальный new(), даже если в классе Screen определена собственная версия:

```
Screen *ps = ::new Screen;
```

Оператор delete(), являющийся членом класса, должен иметь тип void, а в качестве первого параметра принимать void*. Вот как выглядит его объявление

```
class Screen {
public:
   void operator delete( void * );
```

для Screen:

```
};
```

Когда операндом delete служит указатель на объект типа класса, компилятор проверяет, определен ли в этом классе оператор delete(). Если да, то для освобождения памяти вызывается именно он, в противном случае – глобальная версия оператора. Следующая инструкция

```
delete ps;
```

освобождает память, занятую объектом класса Screen, на который указывает ps. Поскольку в Screen есть оператор-член delete(), то применяется именно он. Параметр оператора типа void* автоматически инициализируется значением ps.

Добавление delete() в класс или его удаление оттуда никак не сказываются на пользовательском коде. Вызов delete выглядит одинаково как для глобального оператора, так и для оператора-члена. Если бы в классе Screen не было собственного оператора delete(), то обращение осталось бы правильным, только вместо оператора-члена вызывался бы глобальный оператор.

С помощью оператора разрешения глобальной области видимости можно вызвать глобальный delete(), даже если в Screen определена собственная версия:

```
::delete ps;
```

В общем случае используемый оператор delete() должен соответствовать тому оператору new(), с помощью которого была выделена память. Например, если ps указывает на область памяти, выделенную глобальным new(), то для ее освобождения следует использовать глобальный же delete().

Оператор delete(), определенный для типа класса, может содержать два параметра вместо одного. Первый параметр по-прежнему должен иметь тип void*, а второй –

```
class Screen {
public:
   // заменяет
   // void operator delete( void * );
   void operator delete( void *, size_t );
};
```

предопределенный тип size_t (не забудьте включить заголовочный файл <cstddef>):

Если второй параметр есть, компилятор автоматически инициализирует его значением, равным размеру адресованного первым параметром объекта в байтах. (Этот параметр важен в иерархии классов, когда оператор delete() может наследоваться производным классом. Подробнее наследование обсуждается в главе 17.)

Рассмотрим реализацию операторов new() и delete() в классе Screen более детально. В основе нашей стратегии распределения памяти будет лежать связанный список объектов Screen, на начало которого указывает член freeStore. При каждом обращении к оператору-члену new() возвращается следующий объект из списка. При вызове delete() объект возвращается в список. Если при создании нового объекта список, адресованный freeStore, пуст, то вызывается глобальный оператор new(), чтобы получить блок памяти, достаточный для хранения screenChunk объектов класса Screen.

Как screenChunk, так и freeStore представляют интерес только для Screen, поэтому мы сделаем их закрытыми членами. Кроме того, для всех создаваемых объектов нашего класса значения этих членов должны быть одинаковыми, а следовательно, нужно объявить их статическими. Чтобы поддержать структуру связанного списка объектов

```
class Screen {
public:
   void *operator new( size_t );
   void operator delete( void *, size_t );
   // ...
private:
   Screen *next;
   static Screen *freeStore;
   static const int screenChunk;
};
```

Screen, нам понадобится третий член next:

Вот одна из возможных реализаций оператора new() для класса Screen:

```
#include "Screen.h"
#include <cstddef>

// статические члены инициализируются
// в исходных файлах программы, а не в заголовочных файлах
Screen *Screen::freeStore = 0;
const int Screen::screenChunk = 24;

void *Screen::operator new( size_t size )
{
   Screen *p;

   if ( !freeStore ) {
      // связанный список пуст: получить новый блок
      // вызывается глобальный оператор new
      size_t chunk = screenChunk * size;
      freeStore = p =
         reinterpret_cast< Screen* >( new char[ chunk ] );

      // включить полученный блок в список
      for ( ;
            p != &freeStore[ screenChunk - 1 ];
            ++p )
          p->next = p+1;
      p->next = 0;
    }

    p = freeStore;
    freeStore = freeStore->next;
    return p;
}
```

```
void Screen::operator delete( void *p, size_t )
{
   // вставить "удаленный" объект назад,
   // в список свободных

   ( static_cast< Screen* >( p ) )->next = freeStore;
   freeStore = static_cast< Screen* >( p );
}
```

А вот реализация оператора `delete()`:

Оператор `new()` можно объявить в классе и без соответствующего `delete()`. В таком случае объекты освобождаются с помощью одноименного глобального оператора. Разрешается также объявить и оператор `delete()` без `new()`: объекты будут создаваться с помощью одноименного глобального оператора. Однако обычно эти операторы реализуются одновременно, как в примере выше, поскольку разработчику класса, как правило, нужны оба.

Они являются статическими членами класса, даже если программист явно не объявит их таковыми, и подчиняются обычным ограничениям для подобных функций-членов: им не передается указатель `this`, а следовательно, напрямую они могут получить доступ только к статическим членам. (См. обсуждение статических функций-членов в разделе 13.5.) Причина, по которой эти операторы делаются статическими, заключается в том,

что они вызываются либо перед конструированием объекта класса (new()), либо после его уничтожения (delete()).

Выделение памяти с помощью оператора new(), например:

```
Screen *ptr = new Screen( 10, 20 );
```

```
// Псевдокод на C++
ptr = Screen::operator new( sizeof( Screen ) );
```

эквивалентно последовательному выполнению таких инструкций:

```
Screen::Screen( ptr, 10, 20 );
```

Иными словами, сначала вызывается определенный в классе оператор new(), чтобы выделить память для объекта, а затем этот объект инициализируется конструктором. Если new() неудачно завершает работу, то возбуждается исключение типа bad_alloc и конструктор не вызывается.

Освобождение памяти с помощью оператора delete(), например:

```
delete ptr;
```

```
// Псевдокод на C++
Screen::~Screen( ptr );
```

эквивалентно последовательному выполнению таких инструкций:

```
Screen::operator delete( ptr, sizeof( *ptr ) );
```

Таким образом, при уничтожении объекта сначала вызывается деструктор класса, а затем определенный в классе оператор delete() для освобождения памяти. Если значение ptr равно 0, то ни деструктор, ни delete() не вызываются.

15.8.1. Операторы new[ ] и delete [ ]

Оператор new(), определенный в предыдущем подразделе, вызывается только при выделении памяти для единичного объекта. Так, в данной инструкции вызывается new()

```
// вызывается Screen::operator new()
```

класса Screen:

```
Screen *ps = new Screen( 24, 80 );
```

тогда как ниже вызывается глобальный оператор new[]() для выделения из хипа памяти

```
// вызывается Screen::operator new[]()
```

под массив объектов типа Screen:

```
Screen *psa = new Screen[10];
```

В классе можно объявить также операторы `new[]()` и `delete[]()` для работы с массивами.

Оператор-член `new[]()` должен возвращать значение типа `void*` и принимать в качестве

```
class Screen {
public:
   void *operator new[]( size_t );
   // ...
```

первого параметра значение типа `size_t`. Вот его объявление для `Screen`:

```
};
```

Когда с помощью `new` создается массив объектов типа класса, компилятор проверяет, определен ли в классе оператор `new[]()`. Если да, то для выделения памяти под массив вызывается именно он, в противном случае – глобальный `new[]()`. В следующей инструкции в хипе создается массив из десяти объектов `Screen`:

```
Screen *ps = new Screen[10];
```

В этом классе есть оператор `new[]()`, поэтому он и вызывается для выделения памяти. Его параметр `size_t` автоматически инициализируется значением, равным объему памяти в байтах, необходимому для размещения десяти объектов `Screen`.

Даже если в классе имеется оператор-член `new[]()`, программист может вызвать для создания массива глобальный `new[]()`, воспользовавшись оператором разрешения глобальной области видимости:

```
Screen *ps = ::new Screen[10];
```

Оператор `delete()`, являющийся членом класса, должен иметь тип `void`, а в качестве

```
class Screen {
public:
   void operator delete[]( void * );
```

первого параметра принимать `void*`. Вот как выглядит его объявление для `Screen`:

```
};
```

Чтобы удалить массив объектов класса, `delete` должен вызываться следующим образом:

```
delete[] ps;
```

Когда операндом `delete` является указатель на объект типа класса, компилятор проверяет, определен ли в этом классе оператор `delete[]()`. Если да, то для освобождения памяти вызывается именно он, в противном случае – его глобальная версия. Параметр типа `void*` автоматически инициализируется значением адреса начала области памяти, в которой размещен массив.

Даже если в классе имеется оператор-член delete[](), программист может вызвать глобальный delete[](), воспользовавшись оператором разрешения глобальной области видимости:

```
::delete[] ps;
```

Добавление операторов new[]() или delete[]() в класс или удаление их оттуда не отражаются на пользовательском коде: вызовы как глобальных операторов, так и операторов-членов выглядят одинаково.

При создании массива сначала вызывается new[]() для выделения необходимой памяти, а затем каждый элемент инициализируется с помощью конструктора по умолчанию. Если у класса есть хотя бы один конструктор, но нет конструктора по умолчанию, то вызов оператора new[]() считается ошибкой. Не существует синтаксической конструкции для задания инициализаторов элементов массива или аргументов конструктора класса при создании массива подобным образом.

При уничтожении массива сначала вызывается деструктор класса для уничтожения элементов, а затем оператор delete[]() – для освобождения всей памяти. При этом важно использовать правильный синтаксис. Если в инструкции

```
delete ps;
```

ps указывает на массив объектов класса, то отсутствие квадратных скобок приведет к вызову деструктора лишь для первого элемента, хотя память будет освобождена полностью.

У оператора-члена delete[]() может быть не один, а два параметра, при этом второй

```
class Screen {
public:
   // заменяет
   // void operator delete[]( void* );
   void operator delete[]( void*, size_t );
};
```

должен иметь тип size_t:

Если второй параметр присутствует, то компилятор автоматически инициализирует его значением, равным объему отведенной под массив памяти в байтах.

15.8.2. Оператор размещения new() и оператор delete()

Оператор-член new() может быть перегружен при условии, что все объявления имеют

```
class Screen {
public:
   void *operator new( size_t );
   void *operator new( size_t, Screen * );
   // ...
```

разные списки параметров. Первый параметр должен иметь тип size_t:

```
};
```

Остальные параметры инициализируются аргументами размещения, заданными при

```
void func( Screen *start ) {
   Screen *ps = new (start) Screen;
   // ...
```

вызове new:

```
}
```

Та часть выражения, которая находится после ключевого слова new и заключена в круглые скобки, представляет аргументы размещения. В примере выше вызывается оператор new(), принимающий два параметра. Первый автоматически инициализируется значением, равным размеру класса Screen в байтах, а второй – значением аргумента размещения start.

Можно также перегружать и оператор-член delete(). Однако такой оператор никогда не вызывается из выражения delete. Перегруженный delete() неявно вызывается компилятором, если конструктор, вызванный при выполнении оператора new (это не опечатка, мы действительно имеем в виду new), возбуждает исключение. Рассмотрим использование delete() более внимательно.

Последовательность действий при вычислении выражения

```
Screen *ps = new ( start ) Screen;
```

такова:

1. Вызывается определенный в классе оператор new(size_t, Screen*).
2. Вызывается конструктор по умолчанию класса Screen для инициализации созданного объекта.

Переменная ps инициализируется адресом нового объекта Screen.

Предположим, что оператор класса new(size_t, Screen*) выделяет память с помощью глобального new(). Как разработчик может гарантировать, что память будет освобождена, если вызванный на шаге 2 конструктор возбуждает исключение? Чтобы защитить пользовательский код от утечки памяти, следует предоставить перегруженный оператор delete(), который вызывается только в подобной ситуации.

Если в классе имеется перегруженный оператор с параметрами, типы которых соответствуют типам параметров new(), то компилятор автоматически вызывает его для освобождения памяти. Предположим, есть следующее выражение с оператором размещения new:

```
Screen *ps = new (start) Screen;
```

Если конструктор по умолчанию класса Screen возбуждает исключение, то компилятор ищет delete() в области видимости Screen. Чтобы такой оператор был найден, типы его параметров должны соответствовать типам параметров вызванного new(). Поскольку первый параметр new() всегда имеет тип size_t, а оператора delete() – void*, то первые параметры при сравнении не учитываются. Компилятор ищет в классе Screen оператор delete() следующего вида:

```
void operator delete( void*, Screen* );
```

Если такой оператор будет найден, то он вызывается для освобождения памяти в случае, когда new( ) возбуждает исключение. (Иначе – не вызывается.)

Разработчик класса принимает решение, предоставлять ли delete( ), соответствующий некоторому new( ), в зависимости от того, выделяет ли этот оператор new( ) память самостоятельно или пользуется уже выделенной. В первом случае delete( ) необходимо включить для освобождения памяти, если конструктор возбудит исключение; иначе в нем нет необходимости.

Можно также перегрузить оператор размещения new[]( ) и оператор delete[]( ) для

```
class Screen {
public:
   void *operator new[]( size_t );
   void *operator new[]( size_t, Screen* );
   void operator delete[]( void*, size_t );
   void operator delete[]( void*, Screen* );
   // ...
```

массивов:

```
};
```

Оператор new[]( ) используется в случае, когда в выражении, содержащем new для

```
void func( Screen *start ) {
   // вызывается Screen::operator new[]( size_t, Screen* )
   Screen *ps = new (start) Screen[10];
   // ...
```

распределения массива, заданы соответствующие аргументы размещения:

```
}
```

Если при работе оператора new конструктор возбуждает исключение, то автоматически вызывается соответствующий delete[]( ).

Упражнение 15.9

```
class iStack {
public:
   iStack( int capacity )
         : _stack( capacity ), _top( 0 ) {}
   // ...
private:
   int _top;
   vatcor< int > _stack;
```

Объясните, какие из приведенных инициализаций ошибочны:

```
};
```

```
(a) iStack *ps = new iStack(20);
(b) iStack *ps2 = new const iStack(15);

(c) iStack *ps3 = new iStack[ 100 ];
```

Упражнение 15.10

```
class Exercise {
public:
   Exercise();
   ~Exercise();
};

Exercise *pe = new Exercise[20];
```

Что происходит в следующих выражениях, содержащих `new` и `delete`?

```
delete[] ps;
```

Измените эти выражения так, чтобы вызывались глобальные операторы `new()` и `delete()`.

Упражнение 15.11

Объясните, зачем разработчик класса должен предоставлять оператор `delete()`.

15.9. Определенные пользователем преобразования

Мы уже видели, как преобразования типов применяются к операндам встроенных типов: в разделе 4.14 этот вопрос рассматривался на примере операндов встроенных операторов, а в разделе 9.3 – на примере фактических аргументов вызванной функции для приведения их к типам формальных параметров. Рассмотрим с этой точки зрения

```
char ch; short sh;, int ival;

/* в каждой операции один операнд
 * требует преобразования типа */

ch + ival;          ival + ch;
ch + sh;            ch + ch;
```

следующие шесть операций сложения:

```
ival + sh;          sh + ival;
```

Операнды `ch` и `sh` расширяются до типа `int`. При выполнении операции складываются два значения типа `int`. Расширение типа неявно выполняется компилятором и для пользователя прозрачно.

В этом разделе мы рассмотрим, как разработчик может определить собственные преобразования для объектов типа класса. Такие определенные пользователем преобразования также автоматически вызываются компилятором по мере необходимости. Чтобы показать, зачем они нужны, обратимся снова к классу `SmallInt`, введенному в разделе 10.9.

Напомним, что SmallInt позволяет определять объекты, способные хранить значения из того же диапазона, что unsigned char, т.е. от 0 до 255, и перехватывает ошибки выхода за его границы. Во всех остальных отношениях этот класс ведет себя точно так же, как unsigned char.

Чтобы иметь возможность складывать объекты SmallInt с другими объектами того же класса или со значениями встроенных типов, а также вычитать их, реализуем шесть

```
class SmallInt {
   friend operator+( const SmallInt &, int );
   friend operator-( const SmallInt &, int );
   friend operator-( int, const SmallInt & );
   friend operator+( int, const SmallInt & );
public:
   SmallInt( int ival ) : value( ival ) { }
   operator+( const SmallInt & );
   operator-( const SmallInt & );
   // ...
private:
   int value;
};
```

операторных функций:

Операторы-члены дают возможность складывать и вычитать два объекта SmallInt. Глобальные же операторы-друзья позволяют производить эти операции над объектами данного класса и объектами встроенных арифметических типов. Необходимо только шесть операторов, поскольку любой встроенный арифметический тип может быть

```
SmallInt si( 3 );
```

приведен к типу int. Например, выражение

```
si + 3.14159
```

разрешается в два шага:

1. Константа 3.14159 типа double преобразуется в целое число 3.
2. Вызывается operator+(const SmallInt &, int), который возвращает значение 6.

Если мы хотим поддержать битовые и логические операции, а также операции сравнения и составные операторы присваивания, то сколько же необходимо перегрузить операторов? Сразу и не сосчитаешь. Значительно удобнее автоматически преобразовать объект класса SmallInt в объект типа int.

В языке C++ имеется механизм, позволяющий в любом классе задать набор преобразований, применимых к его объектам. Для SmallInt мы определим приведение объекта к типу int. Вот его реализация:

```
class SmallInt {
public:
   SmallInt( int ival ) : value( ival ) { }

   // конвертер
   // SmallInt ==> int
   operator int() { return value; }

   // перегруженные операторы не нужны

private:
   int value;

};
```

Оператор `int()` – это *конвертер*, реализующий *определенное пользователем преобразование*, в данном случае приведение типа класса к заданному типу `int`. Определение конвертера описывает, что означает преобразование и какие действия компилятор должен выполнить для его применения. Для объекта `SmallInt` смысл преобразования в `int` заключается в том, чтобы вернуть число типа `int`, хранящееся в члене `value`.

Теперь объект класса `SmallInt` можно использовать всюду, где допустимо использование `int`. Если предположить, что перегруженных операторов больше нет и в

```
SmallInt si( 3 );
```

`SmallInt` определен конвертер в `int`, операция сложения

```
si + 3.14159
```

разрешается двумя шагами:

1. Вызывается конвертер класса `SmallInt`, который возвращает целое число 3.
2. Целое число 3 расширяется до 3.0 и складывается с константой двойной точности 3.14159, что дает 6.14159.

Такое поведение больше соответствует поведению операндов встроенных типов по сравнению с определенными ранее перегруженными операторами. Когда значение типа `int` складывается со значением типа `double`, то выполняется сложение двух чисел типа `double` (поскольку тип `int` расширяется до `double`) и результатом будет число того же типа.

В этой программе иллюстрируется применение класса `SmallInt`:

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

int main() {
   cout << "Введите SmallInt, пожалуйста: ";
   while ( cin >> si1 ) {
      cout << "Прочитано значение "
           << si1 << "\nОно ";

      // SmallInt::operator int() вызывается дважды
      cout << ( ( si1 > 127 )
              ? "больше, чем "
              : ( ( si1 < 127 )
                ? "меньше, чем "
                : "равно ") ) << "127\n";

      cout << "\Введите SmallInt, пожалуйста \
              (ctrl-d для выхода): ";
   }
   cout <<"До встречи\n";
}
```

Откомпилированная программа выдает следующие результаты:

```
Введите SmallInt, пожалуйста: 127

Прочитано значение 127
Оно равно 127

Введите SmallInt, пожалуйста (ctrl-d для выхода): 126
Оно меньше, чем 127

Введите SmallInt, пожалуйста (ctrl-d для выхода): 128
Оно больше, чем 127

Введите SmallInt, пожалуйста (ctrl-d для выхода): 256
*** Ошибка диапазона SmallInt: 256 ***
```

```
#include <iostream>

class SmallInt {
   friend istream&
      operator>>( istream &is, SmallInt &s );
   friend ostream&
      operator<<( ostream &is, const SmallInt &s )
      { return os << s.value; }
public:
   SmallInt( int i=0 ) : value( rangeCheck( i ) ){}
   int operator=( int i )
      { return( value = rangeCheck( i ) ); }
   operator int() { return value; }
private:
   int rangeCheck( int );
   int value;
};
```

В реализацию класса `SmallInt` добавили поддержку новой функциональности:

```
istream& operator>>( istream &is, SmallInt &si ) {
   int ix;
   is >> ix;
   si = ix;       // SmallInt::operator=(int)
   return is;
}
int SmallInt::rangeCheck( int i )
{
/* если установлен хотя бы один бит, кроме первых восьми,
 * то значение слишком велико; сообщить и сразу выйти */

   if ( i & ~0377 ) {
      cerr << "\n*** Ошибка диапазона SmallInt: "
           << i << " ***" << endl;
      exit( -1 );
   }
   return i;
```

Ниже приведены определения функций-членов, находящиеся вне тела класса:

```
}
```

15.9.1. Конвертеры

Конвертер – это особый случай функции-члена класса, реализующий определенное пользователем преобразование объекта в некоторый другой тип. Конвертер объявляется в теле класса путем указания ключевого слова `operator`, за которым следует целевой тип преобразования.

Имя, находящееся за ключевым словом, не обязательно должно быть именем одного из встроенных типов. В показанном ниже классе `Token` определено несколько конвертеров. В одном из них для задания имени типа используется `typedef tName`, а в другом – тип

```
#include "SmallInt.h"

typedef char *tName;
class Token {
public:
   Token( char *, int );
   operator SmallInt() { return val; }
   operator tName()    { return name; }
   operator int()      { return val; }
   // другие открытые члены
private:
   SmallInt val;
   char *name;
```

класса `SmallInt`.

```
};
```

Обратите внимание, что определения конвертеров в типы `SmallInt` и `int` одинаковы. Конвертер `Token::operator int()` возвращает значение члена `val`. Поскольку `val` имеет тип `SmallInt`, то неявно применяется `SmallInt::operator int()` для преобразования `val` в тип `int`. Сам `Token::operator int()` неявно употребляется компилятором для преобразования объекта типа `Token` в значение типа `int`. Например,

этот конвертер используется для неявного приведения фактических аргументов t1 и t2

```
#include "Token.h"

void print( int i )
{
   cout << "print( int ) : " << i << endl;
}

Token t1( "integer constant", 127 );
Token t2( "friend", 255 );

int main()
{
   print( t1 );    // t1.operator int()
   print( t2 );    // t2.operator int()
   return 0;
```

типа Token к типу int формального параметра функции print():

```
}
```

```
print( int ) : 127
```

После компиляции и запуска программа выведет такие строки:

```
print( int ) : 255
```

Общий вид конвертера следующий:

```
operator type();
```

где type может быть встроенным типом, типом класса или именем typedef. Конвертеры, в которых type – тип массива или функции, не допускаются. Конвертер должен быть функцией-членом. В его объявлении не должны задаваться ни тип возвращаемого

```
operator int( SmallInt & );  // ошибка: не член

class SmallInt {
public:
   int operator int();       // ошибка: задан тип возвращаемого значения
   operator int( int = 0 );  // ошибка: задан список параметров
   // ...
```

значения, ни список параметров:

```
};
```

Конвертер вызывается в результате явного преобразования типов. Если преобразуемое значение имеет тип класса, у которого есть конвертер, и в операции приведения указан тип этого конвертера, то он и вызывается:

```
#include "Token.h"
Token tok( "function", 78 );

// функциональная нотация: вызывается Token::operator SmallInt()
SmallInt tokVal = SmallInt( tok );
// static_cast: вызывается Token::operator tName()

char *tokName = static_cast< char * >( tok );
```

У конвертера `Token::operator tName()` может быть нежелательный побочный эффект. Попытка прямого обращения к закрытому члену `Token::name` помечается компилятором как ошибка:

```
char *tokName = tok.name;  // ошибка: Token::name - закрытый член
```

Однако наш конвертер, разрешая пользователям непосредственно изменять `Token::name`, делает как раз то, от чего мы хотели защититься. Скорее всего, это не годится. Вот,

```
#include "Token.h"
Token tok( "function", 78 );

char *tokName = tok;   // правильно: неявное преобразование
```

например, как могла бы произойти такая модификация:

```
*tokname = 'P';  // но теперь в члене name находится Punction!
```

Мы намереваемся разрешить доступ к преобразованному объекту класса `Token` только

```
typedef const char *cchar;
class Token {
public:
   operator cchar() { return name; }
   // ...
};

// ошибка: преобразование char* в const char* не допускается
char *pn = tok;
```

для чтения. Следовательно, конвертер должен возвращать тип `const char*`:

```
const char *pn2 = tok;   // правильно
```

Другое решение – заменить в определении `Token` тип `char*` на тип `string` из стандартной библиотеки C++:

```
class Token {
public:
   Token( string, int );
   operator SmallInt() { return val; }
   operator string()   { return name; }
   operator int()      { return val; }
   // другие открытые члены
private:
   SmallInt val;
   string name;
};
```

Семантика конвертера `Token::operator string()` состоит в возврате копии значения (а не указателя на значение) строки, представляющей имя лексемы. Это предотвращает случайную модификацию закрытого члена `name` класса `Token`.

Должен ли целевой тип точно соответствовать типу конвертера? Например, будет ли в

```
extern void calc( double );
Token tok( "constant", 44 );

// Вызывается ли оператор int()? Да
// применяется стандартное преобразование int --> double
```

следующем коде вызван конвертер `int()`, определенный в классе `Token`?

```
calc( tok );
```

Если целевой тип (в данном случае `double`) не точно соответствует типу конвертера (в нашем случае `int`), то конвертер все равно будет вызван при условии, что существует последовательность стандартных преобразований, приводящая к целевому типу из типа конвертера. (Эти последовательности описаны в разделе 9.3.) При обращении к функции `calc()` вызывается `Token::operator int()` для преобразования `tok` из типа `Token` в тип `int`. Затем для приведения результата от типа `int` к типу `double` применяется стандартное преобразование.

Вслед за определенным пользователем преобразованием допускаются только стандартные. Если для достижения целевого типа необходимо еще одно пользовательское преобразование, то компилятор не применяет никаких преобразований. Предположим, что в классе `Token` не определен `operator int()`, тогда следующий вызов будет

```
extern void calc( int );
Token tok( "pointer", 37 );

// если Token::operator int() не определен,
// то этот вызов приводит к ошибке компиляции
```

ошибочным:

```
calc( tok );
```

Если конвертер `Token::operator int()` не определен, то приведение `tok` к типу `int` потребовало бы вызова двух определенных пользователем конвертеров. Сначала фактический аргумент `tok` надо было бы преобразовать из типа `Token` в тип `SmallInt` с помощью конвертера

```
Token::operator SmallInt()
```

а затем результат привести к типу `int` – тоже с помощью пользовательского конвертера

```
Token::operator int()
```

Вызов `calc(tok)` помечается компилятором как ошибка, так как не существует неявного преобразования из типа `Token` в тип `int`.

Если логического соответствия между типом конвертера и типом класса нет, назначение

```
class Date {
public:
   // попробуйте догадаться, какой именно член возвращается!
   operator int();
private:
   int month, day, year;
```

конвертера может оказаться непонятным читателю программы:

```
};
```

Какое значение должен вернуть конвертер `int()` класса `Date`? Сколь бы основательными ни были причины для того или иного решения, читатель останется в недоумении относительно того, как пользоваться объектами класса `Date`, поскольку между ними и целыми числами нет явного логического соответствия. В таких случаях лучше вообще *не* определять конвертер.

15.9.2. Конструктор как конвертер

Набор конструкторов класса, принимающих единственный параметр, например, `SmallInt(int)` класса `SmallInt`, определяет множество неявных преобразований в значения типа `SmallInt`. Так, конструктор `SmallInt(int)` преобразует значения типа

```
extern void calc( SmallInt );
int i;

// необходимо преобразовать i в значение типа SmallInt
// это достигается применением SmallInt(int)
```

`int` в значения типа `SmallInt`.

```
calc( i );
```

При вызове `calc(i)` число `i` преобразуется в значение типа `SmallInt` с помощью конструктора `SmallInt(int)`, вызванного компилятором для создания временного объекта нужного типа. Затем копия этого объекта передается в `calc()`, как если бы вызов функции был записан в форме:

```
// Псевдокод на C++
// создается временный объект типа SmallInt
{
   SmallInt temp = SmallInt( i );
   calc( temp );
}
```

Фигурные скобки в этом примере обозначают время жизни данного объекта: он уничтожается при выходе из функции.

```
class Number {
public:
   // создание значения типа Number из значения типа SmallInt
   Number( const SmallInt & );
   // ...
};
```

Типом параметра конструктора может быть тип некоторого класса:

В таком случае значение типа SmallInt можно использовать всюду, где допустимо

```
extern void func( Number );
SmallInt si(87);

int main()
{  // вызывается Number( const SmallInt & )
   func( si );
   // ...
}
```

значение типа Number:

Если конструктор используется для выполнения неявного преобразования, то должен ли тип его параметра точно соответствовать типу подлежащего преобразованию значения? Например, будет ли в следующем коде вызван SmallInt(int), определенный в классе

```
extern void calc( SmallInt );
double dobj;

// вызывается ли SmallInt(int)? Да
// dobj преобразуется приводится от double к int
// стандартным преобразованием
```

SmallInt, для приведения dobj к типу SmallInt?

```
calc( dobj );
```

Если необходимо, к фактическому аргументу применяется последовательность стандартных преобразований до того, как вызвать конструктор, выполняющий определенное пользователем преобразование. При обращении к функции calc()употребляется стандартное преобразование dobj из типа double в тип int. Затем уже для приведения результата к типу SmallInt вызывается SmallInt(int).

Компилятор неявно использует конструктор с единственным параметром для преобразования его типа в тип класса, к которому принадлежит конструктор. Однако иногда удобнее, чтобы конструктор `Number(const SmallInt&)` можно было вызывать только для инициализации объекта типа `Number` значением типа `SmallInt`, но ни в коем случае не для выполнения неявных преобразований. Чтобы избежать такого

```
class Number {
public:
   // никогда не использовать для неявных преобразований
   explicit Number( const SmallInt & );
   // ...
```

употребления конструктора, объявим его явным (`explicit`):

```
};
```

Компилятор никогда не применяет явные конструкторы для выполнения неявных

```
extern void func( Number );
SmallInt si(87);

int main()
{  // ошибка: не существует неявного преобразования из SmallInt в Number
   func( si );
   // ...
```

преобразований типов:

```
}
```

Однако такой конструктор все же можно использовать для преобразования типов, если

```
SmallInt si(87);

int main()
{  // ошибка: не существует неявного преобразования из SmallInt в Number
   func( si );
   func( Number( si ) );  // правильно: приведение типа
   func( static_cast< Number >( si ) );  // правильно: приведение типа
```

оно запрошено явно в форме оператора приведения типа:

```
}
```

15.10. Выбор преобразования A

Определенное пользователем преобразование реализуется в виде конвертера или конструктора. Как уже было сказано, после преобразования, выполненного конвертером, разрешается использовать стандартное преобразование для приведения возвращенного значения к целевому типу. Трансформации, выполненной конструктором, также может предшествовать стандартное преобразование для приведения типа аргумента к типу формального параметра конструктора.

Последовательность определенных пользователем преобразований – это комбинация определенного пользователем и стандартного преобразования, которая необходима для приведения значения к целевому типу. Такая последовательность имеет вид:

Последовательность стандартных преобразований ->

Определенное пользователем преобразование ->

Последовательность стандартных преобразований

где определенное пользователем преобразование реализуется конвертером либо конструктором.

Не исключено, что для трансформации исходного значения в целевой тип существует две разных последовательности пользовательских преобразований, и тогда компилятор должен выбрать из них лучшую. Рассмотрим, как это делается.

В классе разрешается определять много конвертеров. Например, в нашем классе `Number` их два: `operator int()` и `operator float()`, причем оба способны преобразовать объект типа `Number` в значение типа `float`. Естественно, можно воспользоваться конвертером `Token::operator float()` для прямой трансформации. Но и `Token::operator int()` тоже подходит, так как результат его применения имеет тип `int` и, следовательно, может быть преобразован в тип `float` с помощью стандартного преобразования. Является ли трансформация неоднозначной, если имеется несколько

```
class Number {
public:
   operator float();
   operator int();
   // ...
};
Number num;
```

таких последовательностей? Или какую-то из них можно предпочесть остальным?

```
float ff = num;   // какой конвертер? operator float()
```

В таких случаях выбор наилучшей последовательности определенных пользователем преобразований основан на анализе последовательности преобразований, которая применяется после конвертера. В предыдущем примере можно применить такие две последовательности:

1. `operator float()` -> точное соответствие
2. `operator int()` -> стандартное преобразование

Как было сказано в разделе 9.3, точное соответствие лучше стандартного преобразования. Поэтому первая последовательность лучше второй, а значит, выбирается конвертер `Token::operator float()`.

Может случиться так, что для преобразования значения в целевой тип применимы два разных конструктора. В этом случае анализируется последовательность стандартных преобразований, предшествующая вызову конструктора:

```
class SmallInt {
public:
   SmallInt( int ival ) : value( ival ) { }
   SmallInt( double dval )
           : value( static_cast< int >( dval ) );
   { }
};

extern void manip( const SmallInt & );

int main() {
   double dobj;
   manip( dobj );   // правильно: SmallInt( double )
}
```

Здесь в классе SmallInt определено два конструктора – SmallInt(int) и SmallInt(double), которые можно использовать для изменения значения типа double в объект типа SmallInt: SmallInt(double) трансформирует double в SmallInt напрямую, а SmallInt(int) работает с результатом стандартного преобразования double в int. Таким образом, имеются две последовательности определенных пользователем преобразований:

1. точное соответствие -> SmallInt(double)
2. стандартное преобразование -> SmallInt(int)

Поскольку точное соответствие лучше стандартного преобразования, то выбирается конструктор SmallInt(double).

Не всегда удается решить, какая последовательность лучше. Может случиться, что все они одинаково хороши, и тогда мы говорим, что преобразование *неоднозначно*. В таком случае компилятор не применяет никаких неявных трансформаций. Например, если в

```
class Number {
public:
   operator float();
   operator int();
   // ...
};
```

классе Number есть два конвертера:

```
// ошибка: можно применить как float(), так и int()
```

то невозможно неявно преобразовать объект типа Number в тип long. Следующая инструкция вызывает ошибку компиляции, так как выбор последовательности

```
long lval = num;
```

определенных пользователем преобразований неоднозначен:

Для трансформации num в значение типа long применимы две такие последовательности:

1. operator float() -> стандартное преобразование
2. operator int() -> стандартное преобразование

Поскольку в обоих случаях за использованием конвертера следует применение стандартного преобразования, то обе последовательности одинаково хороши и компилятор не может выбрать ни одну из них.

```
// правильно: явное приведение типа
```

С помощью явного приведения типов программист способен задать нужное изменение:

```
long lval = static_cast< int >( num );
```

Вследствие такого указания выбирается конвертер `Token::operator int()`, за которым следует стандартное преобразование в `long`.

Неоднозначность при выборе последовательности трансформаций может возникнуть и

```
class SmallInt {
public:
   SmallInt( const Number & );
   // ...
};

class Number {
public:
   operator SmallInt();
   // ...
};

extern void compute( SmallInt );
extern Number num;
```

тогда, когда два класса определяют преобразования друг в друга. Например:

```
compute( num );  // ошибка: возможно два преобразования
```

Аргумент `num` преобразуется в тип `SmallInt` двумя разными способами: с помощью конструктора `SmallInt::SmallInt(const Number&)` либо с помощью конвертера `Number::operator SmallInt()`. Поскольку оба изменения одинаково хороши, вызов считается ошибкой.

Для разрешения неоднозначности программист может явно вызвать конвертер класса

```
// правильно: явный вызов устраняет неоднозначность
```

`Number`:

```
compute( num.operator SmallInt() );
```

Однако для разрешения неоднозначности не следует использовать явное приведение типов, поскольку при отборе преобразований, подходящих для приведения типов, рассматриваются как конвертер, так и конструктор:

```
compute( SmallInt( num ) );  // ошибка: по-прежнему неоднозначно
```

Как видите, наличие большого числа подобных конвертеров и конструкторов небезопасно, поэтому их. следует применять с осторожностью. Ограничить

использование конструкторов при выполнении неявных преобразований (а значит, уменьшить вероятность неожиданных эффектов) можно путем объявления их явными.

15.10.1. Еще раз о разрешении перегрузки функций

В главе 9 подробно описывалось, как разрешается вызов перегруженной функции. Если фактические аргументы при вызове имеют тип класса, указателя на тип класса или указателя на члены класса, то на роль возможных кандидатов претендует большее число функций. Следовательно, наличие таких аргументов оказывает влияние на первый шаг процедуры разрешения перегрузки – отбор множества функций-кандидатов.

На третьем шаге этой процедуры выбирается наилучшее соответствие. При этом ранжируются преобразования типов фактических аргументов в типы формальных параметров функции. Если аргументы и параметры имеют тип класса, то в множество возможных преобразований следует включать и последовательности определенных пользователем преобразований, также подвергая их ранжированию.

В этом разделе мы детально рассмотрим, как фактические аргументы и формальные параметры типа класса влияют на отбор функций-кандидатов и как последовательности определенных пользователем преобразований сказываются на выборе наилучшей из устоявших функции.

15.10.2. Функции-кандидаты

Функцией-кандидатом называется функция с тем же именем, что и вызванная.

```
SmallInt si(15);
```

Предположим, что имеется такой вызов:

```
add( si, 566 );
```

Функция-кандидат должна иметь имя add. Какие из объявлений add() принимаются во внимание? Те, которые видимы в точке вызова.

Например, обе функции add(), объявленные в глобальной области видимости, будут

```
const matrix& add( const matrix &, int );
double add( double, double );

int main() {
   SmallInt si(15);
   add( si, 566 );
   // ...
```

кандидатами для следующего вызова:

```
}
```

Рассмотрение функций, чьи объявления видны в точке вызова, производится не только для вызовов с аргументами типа класса. Однако для них поиск объявлений проводится еще в двух областях видимости:

- если фактический аргумент – это объект типа класса, указатель или ссылка на тип класса либо указатель на член класса и этот тип объявлен в пользовательском

пространстве имен, то к множеству функций-кандидатов добавляются функции,

```
namespace NS {
   class SmallInt { /* ... */ };
   class String { /* ... */ };
   String add( const String &, const String & );
}

int main() {
   // si имеет тип class SmallInt:
   // класс объявлен в пространстве имен NS
   NS::SmallInt si(15);

   add( si, 566 );  // NS::add() - функция-кандидат
   return 0;
}
```

объявленные в этом же пространстве и имеющие то же имя, что и вызванная:

Аргумент `si` имеет тип `SmallInt`, т.е. тип класса, объявленного в пространстве имен `NS`. Поэтому к множеству функций-кандидатов добавляется `add(const String &, const String &)`, объявленная в этом пространстве имен;

- если фактический аргумент – это объект типа класса, указатель или ссылка на класс либо указатель на член класса и у этого класса есть друзья, имеющие то же имя, что и

```
namespace NS {
   class SmallInt {
      friend SmallInt add( SmallInt, int ) { /* ... */ }
   };
}

int main() {
   NS::SmallInt si(15);

   add( si, 566 );  // функция-друг add() - кандидат
   return 0;
}
```

вызванная функция, то они добавляются к множеству функций-кандидатов:

Аргумент функции `si` имеет тип `SmallInt`. Функция-друг класса `SmallInt` `add(SmallInt, int)` – член пространства имен `NS`, хотя непосредственно в этом пространстве она не объявлена. При обычном поиске в `NS` функция-друг не будет найдена. Однако при вызове `add()` с аргументом типа класса `SmallInt` принимаются во внимание и добавляются к множеству кандидатов также друзья этого класса, объявленные в списке его членов.

Таким образом, если в списке фактических аргументов функции есть объект, указатель или ссылка на класс, а также указатели на члены класса, то множество функций-кандидатов состоит из множества функций, видимых в точке вызова, или объявленных в том же пространстве имен, где определен тип класса, или объявленных друзьями этого класса.

Рассмотрим следующий пример:

```
namespace NS {
   class SmallInt {
      friend SmallInt add( SmallInt, int ) { /* ... */ }
   };
   class String { /* ... */ };
   String add( const String &, const String & );
}

const matrix& add( const matrix &, int );
double add( double, double );

int main() {
   // si имеет тип class SmallInt:
   // класс объявлен в пространстве имен NS
   NS::SmallInt si(15);

   add( si, 566 );  // вызывается функция-друг
   return 0;

}
```

Здесь кандидатами являются:

```
const matrix& add( const matrix &, int )
```

- глобальные функции:

```
double add( double, double )
```

- функция из пространства имен:

```
NS::add( const String &, const String & )
```

- функция-друг:

```
NS::add( SmallInt, int )
```

При разрешении перегрузки выбирается функция-друг класса `SmallInt NS::add( SmallInt, int )` как наилучшая из устоявших: оба фактических аргумента точно соответствуют заданным формальным параметрам.

Разумеется, вызванная функция может быть несколько аргументов типа класса, указателя или ссылки на класс либо указателя на член класса. Допускаются разные типы классов для каждого из таких аргументов. Поиск функций-кандидатов для них ведется в пространстве имен, где определен класс, и среди функций-друзей класса. Поэтому результирующее множество кандидатов для вызова функции с такими аргументами содержит функции из разных пространств имен и функции-друзья, объявленные в разных классах.

15.10.3. Функции-кандидаты для вызова функции в области видимости класса

Когда вызов функции вида

```
calc(t)
```

встречается в области видимости класса (например, внутри функции-члена), то первая часть множества кандидатов, описанного в предыдущем подразделе (т.е. множество, включающее объявления функций, видимых в точке вызова), может содержать не только функции-члены класса. Для построения такого множества применяется разрешение имени. (Эта тема детально разбиралась в разделах 13.9 – 13.12.)

```
namespace NS {
   struct myClass {
      void k( int );
      static void k( char* );
      void mf();
   };
   int k( double );
};

void h(char);

void NS::myClass::mf() {
   h('a');   // вызывается глобальная h( char )
   k(4);     // вызывается myClass::k( int )
}
```

Рассмотрим пример:

Как отмечалось в разделе 13.11, квалификаторы `NS::myClass::` просматриваются в обратном порядке: сначала поиск видимого объявления для имени, использованного в определении функции-члена `mf()`, ведется в классе `myClass`, а затем – в пространстве имен `NS`. Рассмотрим первый вызов:

```
h( 'a' );
```

При разрешении имени `h()` в определении функции-члена `mf()` сначала просматриваются функции-члены `myClass`. Поскольку функции-члена с таким именем в области видимости этого класса нет, то далее поиск идет в пространстве имен `NS`. Функции `h()`нет и там, поэтому мы переходим в глобальную область видимости. Результат – глобальная функция `h(char)`, единственная функция-кандидат, видимая в точке вызова.

Как только найдено подходящее объявление, поиск прекращается. Следовательно, множество содержит только те функции, объявления которых находятся в областях видимости, где разрешение имени завершилось успешно. Это можно наблюдать на примере построения множества кандидатов для вызова

```
k( 4 );
```

Сначала поиск ведется в области видимости класса `myClass`. При этом найдены две функции-члена `k(int)` и `k(char*)`. Поскольку множество кандидатов содержит лишь функции, объявленные в той области, где разрешение успешно завершилось, то пространство имен `NS` не просматривается и функция `k(double)` в данное множество не включается.

Если обнаруживается, что вызов неоднозначен, поскольку в множестве нет наиболее подходящей функции, то компилятор выдает сообщение об ошибке. Поиск кандидатов, лучше соответствующих фактическим аргументам, в объемлющих областях видимости не производится.

15.10.4. Ранжирование последовательностей определенных пользователем преобразований

Фактический аргумент функции может быть неявно приведен к типу формального параметра с помощью последовательности определенных пользователем преобразований. Как это влияет на разрешение перегрузки? Например, если имеется следующий вызов

```
class SmallInt {
public:
   SmallInt( int );
};

extern void calc( double );
extern void calc( SmallInt );
int ival;

int main() {
   calc( ival );   // какая calc() вызывается?
}
```

`calc()`, то какая функция будет вызвана?

Выбирается функция, формальные параметры которой лучше всего соответствуют типам фактических аргументов. Она называется лучшим соответствием или наилучшей из устоявших функций. Для выбора такой функции неявные преобразования, примененные к фактическим аргументам, подвергаются ранжированию. Лучшей из устоявших считается та, для которой примененные к аргументам изменения *не хуже*, чем для любой другой устоявшей, а хотя бы для одного аргумента они *лучше*, чем для всех остальных функций.

Последовательность стандартных преобразований всегда лучше последовательности определенных пользователем преобразований. Так, при вызове `calc()` из примера выше обе функции `calc()` являются устоявшими. `calc(double)` устояла потому, что существует стандартное преобразование типа фактического аргумента `int` в тип формального параметра `double`, а `calc(SmallInt)` – потому, что имеется определенное пользователем преобразование из `int` в `SmallInt`, которое использует конструктор `SmallInt(int)`. Следовательно, наилучшей из устоявших функций будет `calc(double)`.

А как сравниваются две последовательности определенных пользователем преобразований? Если в них используются разные конвертеры или разные конструкторы, то обе такие последовательности считаются одинаково хорошими:

```
class Number {
public:
   operator SmallInt();
   operator int();
   // ...
};

extern void calc( int );
extern void calc( SmallInt );
extern Number num;

calc( num );   // ошибка: неоднозначность
```

Устоявшими окажутся и calc(int), и calc(SmallInt); первая – поскольку конвертер Number::operator int()преобразует фактический аргумент типа Number в формальный параметр типа int, а вторая потому, что конвертер Number::operator SmallInt() преобразует фактический аргумент типа Number в формальный параметр типа SmallInt. Так как последовательности определенных пользователем преобразований всегда имеют одинаковый ранг, то компилятор не может выбрать, какая из них лучше. Таким образом, этот вызов функции неоднозначен и приводит к ошибке компиляции.

```
// явное указание преобразования устраняет неоднозначность
```

Есть способ разрешить неоднозначность, указав преобразование явно:

```
calc( static_cast< int >( num ) );
```

Явное приведение типов заставляет компилятор преобразовать аргумент num в тип int с помощью конвертера Number::operator int(). Фактический аргумент тогда будет иметь тип int, что точно соответствует функции calc(int), которая и выбирается в качестве наилучшей.

Допустим, в классе Number не определен конвертер Number::operator int(). Будет ли

```
// определен только Number::operator SmallInt()
```

тогда вызов

```
calc( num );   // по-прежнему неоднозначен?
```

по-прежнему неоднозначен? Вспомните, что в SmallInt также есть конвертер, способный

```
class SmallInt {
public:
   operator int();
   // ...
```

преобразовать значение типа SmallInt в int.

```
};
```

Можно предположить, что функция calc() вызывается, если сначала преобразовать фактический аргумент num из типа Number в тип SmallInt с помощью конвертера

Number::operator SmallInt(), а затем результат привести к типу int с помощью SmallInt::operator SmallInt(). Однако это не так. Напомним, что в последовательность определенных пользователем преобразований может входит несколько стандартных преобразований, но лишь одно пользовательское. Если конвертер Number::operator int() не определен, то функция calc(int) не считается устоявшей, поскольку не существует неявного преобразования из типа фактического аргумента num в тип формального параметра int.

Поэтому в отсутствие конвертера Number::operator int() единственной устоявшей функцией будет calc(SmallInt), в пользу которой и разрешается вызов.

Если в двух последовательностях определенных пользователем преобразований употребляется один и тот же конвертер, то выбор наилучшей зависит от

```
class SmallInt {
public:
   operator int();
   // ...
```

последовательности стандартных преобразований, выполняемых после его вызова:

```
void manip( int );
void manip( char );

SmallInt si ( 68 );

main() {
   manip( si );    // вызывается manip( int )
};
}
```

Как manip(int), так и manip(char) являются устоявшими функциями; первая – потому, что конвертер SmallInt::operator int() преобразует фактический аргумент типа SmallInt в тип формального параметра int, а вторая – потому, что тот же конвертер преобразует SmallInt в int, после чего результат с помощью стандартного преобразования приводится к типу char. Последовательности определенных

```
manip(int) : operator int()->точное соответствие
```

пользователем преобразований выглядят так:

```
manip(int) : operator int()->стандартное преобразование
```

Поскольку в обеих последовательностях используется один и тот же конвертер, то для определения лучшей из них анализируется ранг последовательности стандартных преобразований. Так как точное соответствие лучше преобразования, то наилучшей из устоявших будет функция manip(int).

Подчеркнем, что такой критерий выбора принимается только тогда, когда в обеих последовательностях определенных пользователем преобразований применяется один и тот же конвертер. Этим наш пример отличается от приведенных в конце раздела 15.9, где мы показывали, как компилятор выбирает пользовательское преобразование некоторого значения в данный целевой тип: исходный и целевой типы были фиксированы, и компилятору приходилось выбирать между различными определенными пользователем

преобразованиями одного типа в другой. Здесь же рассматриваются две разных функции с разными типами формальных параметров, и целевые типы отличаются. Если для двух разных типов параметров нужны различные определенные пользователем преобразования, то предпочесть один тип другому возможно только в том случае, когда в обеих последовательностях используется один и тот же конвертер. Если это не так, то для выбора наилучшего целевого типа оцениваются стандартные преобразования, следующие

```
class SmallInt {
public:
   operator int();
   operator float();
   // ...
```

за применением конвертера. Например:

```
void compute( float );
void compute( char );

SmallInt si ( 68 );

main() {
   compute( si );   // неоднозначность
};
}
```

И `compute(float)`, и `compute(int)` – устоявшие функции. `compute(float)` – потому, что конвертер `SmallInt::operator float()`преобразует аргумент типа `SmallInt` в тип параметра `float`, а `compute(char)` – потому, что `SmallInt::operator int()` преобразует аргумент типа `SmallInt` в тип `int`, после чего результат стандартно

```
compute(float) : operator float()->точное соответствие
```

приводится к типу `char`. Таким образом, имеются последовательности:

```
compute(char) : operator char()->стандартное преобразование
```

Поскольку в них применяются разные конвертеры, то невозможно определить, у какой функции формальные параметры лучше соответствуют вызову. Для выбора лучшей из двух ранг последовательности стандартных преобразований не используется. Вызов помечается компилятором как неоднозначный.

Упражнение 15.12

В классах стандартной библиотеки C++ нет определений конвертеров, а большинство конструкторов, принимающих один параметр, объявлены явными. Однако определено множество перегруженных операторов. Как вы думаете, почему при проектировании было принято такое решение?

Упражнение 15.13

Почему перегруженный оператор ввода для класса `SmallInt`, определенный в начале этого раздела, реализован не так:

```
istream& operator>>( istream &is, SmallInt &si )
{
   return ( is >> is.value );
}
```

Упражнение 15.14

Приведите возможные последовательности определенных пользователем преобразований

```
class LongDouble {
   operator double();
   operator float();
};
```

для следующих инициализаций. Каким будет результат каждой инициализации?

```
(a) int ex1 = ldObj;
extern LongDouble ldObj;

(b) float ex2 = ldObj;
```

Упражнение 15.15

Назовите три множества функций-кандидатов, рассматриваемых при разрешении перегрузки функции в случае, когда хотя бы один ее аргумент имеет тип класса.

Упражнение 15.16

Какая из функций `calc()` выбирается в качестве наилучшей из устоявших в данном случае? Покажите последовательности преобразований, необходимых для вызова каждой

```
class LongDouble {
public:
   LongDouble( double );
   // ...
};

extern void calc( int );
extern void calc( LongDouble );
double dval;

int main() {
   calc( dval );   // какая функция?
}
```

функции, и объясните, почему одна из них лучше другой.

15.11. Разрешение перегрузки и функции-члены A

Функции-члены также могут быть перегружены, и в этом случае тоже применяется процедура разрешения перегрузки для выбора наилучшей из устоявших. Такое разрешение очень похоже на аналогичную процедуру для обычных функций и состоит из тех же трех шагов:

1. Отбор функций-кандидатов.
2. Отбор устоявших функций.
3. Выбор наилучшей из устоявших функции.

Однако есть небольшие различия в алгоритмах формирования множества кандидатов и отбора устоявших функций-членов. Эти различия мы и рассмотрим в настоящем разделе.

15.11.1. Объявления перегруженных функций-членов

```
class myClass {
public:
   void f( double );
char f( char, char );  // перегружает myClass::f( double )
   // ...
```

Функции-члены класса можно перегружать:

```
};
```

Как и в случае функций, объявленных в пространстве имен, функции-члены могут иметь одинаковые имена при условии, что списки их параметров различны либо по числу параметров, либо по их типам. Если же объявления двух функций-членов отличаются только типом возвращаемого значения, то второе объявление считается ошибкой

```
class myClass {
public:
   void mf();
   double mf();   // ошибка: так перегружать нельзя
   // ...
```

компиляции:

```
};
```

В отличие от функций в пространствах имен, функции-члены должны быть объявлены только один раз. Если даже тип возвращаемого значения и списки параметров двух функций-членов совпадают, то второе объявление компилятор трактует как неверное

```
class myClass {
public:
   void mf();
   void mf();   // ошибка: повторное объявление
   // ...
```

повторное объявление:

```
};
```

Все функции из множества перегруженных должны быть объявлены в одной и той же области видимости. Поэтому функции-члены никогда не перегружают функций, объявленных в пространстве имен. Кроме того, поскольку у каждого класса своя область видимости, функции, являющиеся членами разных классов, не перегружают друг друга.

Множество перегруженных функций-членов может содержать как статические, так и

```
class myClass {
public:
   void mcf( double );
   static void mcf( int* );   // перегружает myClass::mcf( double )
    // ...
```

нестатические функции:

```
};
```

Какая из функций-членов будет вызвана – статическая или нестатическая – зависит от результатов разрешения перегрузки. Процесс разрешения в ситуации, когда устояли как статические, так и нестатические члены, мы подробно рассмотрим в следующем разделе.

15.11.2. Функции-кандидаты

```
mc.mf( arg );
```

Рассмотрим два вида вызовов функции-члена:

```
pmc->mf( arg );
```

где `mc` – выражение типа `myClass`, а `pmc` – выражение типа “указатель на тип `myClass`”. Множество кандидатов для обоих вызовов составлено из функций, найденных в области видимости класса `myClass` при поиске объявления `mf()`.

Аналогично для вызова функции вида

```
myClass::mf( arg );
```

множество кандидатов также состоит из функций, найденных в области видимости класса `myClass` при поиске объявления `mf()`. Например:

```
class myClass {
public:
   void mf( double );
   void mf( char, char = '\n' );
   static void mf( int* );
   // ...
};

int main() {
   myClass mc;
   int iobj;
   mc.mf( iobj );
}
```

Кандидатами для вызова функции в `main()` являются все три функции-члена `mf()`,

```
void mf( double );
void mf( char, char = '\n' );
```

объявленные в `myClass`:

```
static void mf( int* );
```

Если бы в `myClass` не было объявлено ни одной функции-члена с именем `mf()`, то множество кандидатов оказалось бы пустым. (На самом деле рассматривались бы также и функции из базовых классов. О том, как они попадают в это множество, мы поговорим в разделе 19.3.) Если для вызова функции не оказывается кандидатов, компилятор выдает сообщение об ошибке.

15.11.3. Устоявшие функции

Устоявшей называется функция из множества кандидатов, которая может быть вызвана с данными фактическими аргументами. Чтобы она устояла, должны существовать неявные преобразования между типами фактических аргументов и формальных параметров.

```
class myClass {
public:
   void mf( double );
   void mf( char, char = '\n' );
   static void mf( int* );
   // ...
};

int main() {
   myClass mc;
   int iobj;
   mc.mf( iobj );  // какая именно функция-член mf()? Неоднозначно
```

Например:

```
}
```

В этом фрагменте для вызова `mf()` из `main()` есть две устоявшие функции:

```
void mf( double );
void mf( char, char = '\n' );
```

- `mf(double)` устояла потому, что у нее только один параметр и существует стандартное преобразование аргумента `iobj` типа `int` в параметр типа `double`;
- `mf(char,char)` устояла потому, что для второго параметра имеется значение по умолчанию и существует стандартное преобразование аргумента `iobj` типа `int` в тип `char` первого формального параметра.

При выборе наилучшей из устоявших функции преобразования типов, применяемые к каждому фактическому аргументу, ранжируются. Лучшей считается та, для которое все использованные преобразования *не хуже*, чем для любой другой устоявшей функции, и хотя бы для одного аргумента такое преобразование *лучше*, чем для всех остальных функций.

В предыдущем примере в каждой из двух устоявших функций для приведения типа фактического аргумента к типу формального параметра применено стандартное преобразование. Вызов считается неоднозначным, так как обе функции-члена разрешают его одинаково хорошо.

Независимо от вида вызова функции, в множество устоявших могут быть включены как

```
class myClass {
public:
   static void mf( int );
   char mf( char );
};

int main() {
   char cobj;
   myClass::mf( cobj );  // какая именно функция-член?
}
```

статические, так и нестатические члены:

Здесь функция-член `mf()` вызывается с указанием имени класса и оператора разрешения области видимости `myClass::mf()`. Однако не задан ни объект (с оператором “точка”), ни указатель на объект (с оператором “стрелка”). Несмотря на это, нестатическая функция-член `mf(char)` все же включается в множество устоявших наряду со статическим членом `mf(int)`.

Затем процесс разрешения перегрузки продолжается: на основе ранжирования преобразований типов, примененных к фактическим аргументам, чтобы выбрать наилучшую из устоявших функций. Аргумент `cobj` типа `char` точно соответствует формальному параметру `mf(char)` и может быть расширен до типа формального параметра `mf(int)`. Поскольку ранг точного соответствия выше, то выбирается функция `mf(char)`.

Однако эта функция-член не является статической и, следовательно, вызывается только через объект или указатель на объект класса `myClass` с помощью одного из операторов доступа. В такой ситуации, если объект не указан и, значит, вызов функции невозможен (как раз наш случай), компилятор считает его ошибкой.

Еще одна особенность функций-членов, которую надо принимать во внимание при формировании множества устоявших функций, – это наличие спецификаторов const или volatile у нестатических членов. (Они рассматривались в разделе 13.3.) Как они влияют на процесс разрешения перегрузки? Пусть в классе myClass есть следующие

```
class myClass {
public:
   static void mf( int* );
   void mf( double );
   void mf( int ) const;
   // ...
```

функции-члены:

```
};
```

Тогда и статическая функция-член mf(int*), и константная функция mf(int), и неконстантная функция mf(double) включаются в множество кандидатов для

```
int main() {
   const myClass mc;
   double dobj;
   mc.mf( dobj );   // какая из функций-членов mf()?
```

показанного ниже вызова. Но какие из них войдут в множество устоявших?

```
}
```

Исследуя преобразования, которые надо применить к фактическим аргументам, мы обнаруживаем, что устояли функции mf(double) и mf(int). Тип double фактического аргумента dobj точно соответствует типу формального параметра mf(double) и может быть приведен к типу параметра mf(int) с помощью стандартного преобразования.

Если при вызове функции-члена используются операторы доступа “точка” или “стрелка”, то при отборе функций в множество устоявших принимается во внимание тип объекта или указателя, для которого вызвана функция.

mc – это константный объект, для которого можно вызывать только нестатические константные функции-члены. Следовательно, неконстантная функция-член mf(double) исключается из множества устоявших, и остается в нем единственная функция mf(int), которая и вызывается.

А если константный объект использован для вызова статической функции-члена? Ведь для такой функции нельзя задавать спецификатор const или volatile, так можно ли ее

```
class myClass {
public:
   static void mf( int );
   char mf( char );
};
int main() {
   const myClass mc;
   int iobj;
   mc.mf( iobj );   // можно ли вызывать статическую функцию-член?
```

вызывать через константный объект?

```
}
```

Статические функции-члены являются общими для всех объектов одного класса. Напрямую они могут обращаться только к статическим членам класса. Так, нестатические члены константного объекта `mc` недоступны статической `mf(int)`. По этой причине разрешается вызывать статическую функцию-член для константного объекта с помощью операторов “точка” или “стрелка”.

Таким образом, статические функции-члены не исключаются из множества устоявших и при наличии спецификаторов `const` или `volatile` у объекта, для которого они вызваны. Статические функции-члены рассматриваются как соответствующие любому объекту или указателю на объект своего класса.

В примере выше `mc` – константный объект, поэтому функция-член `mf(char)` исключается из множества устоявших. Но функция-член `mf(int)` в нем остается, так как является статической. Поскольку это единственная устоявшая функция, она и оказывается наилучшей.

15.12. Разрешение перегрузки и операторы A

В классах могут быть объявлены перегруженные операторы и конвертеры. Предположим,

```
SomeClass sc;
```

при инициализации встретился оператор сложения:

```
int iobj = sc + 3;
```

Как компилятор решает, что следует сделать: вызвать перегруженный оператор для класса `SomeClass` или конвертировать операнд `sc` во встроенный тип, а затем уже воспользоваться встроенным оператором?

Ответ зависит от множества перегруженных операторов и конвертеров, определенных в `SomeClass`. При выборе оператора для выполнения сложения применяется процесс разрешения перегрузки функции. В данном разделе мы расскажем, как этот процесс позволяет выбрать нужный оператор, когда операндами являются объекты типа класса.

При разрешении перегрузки используется все та же процедура из трех шагов, представленная в разделе 9.2:

1. Отбор функций-кандидатов.
2. Отбор устоявших функций.
3. Выбор наилучшей из устоявших функции.

Рассмотрим эти шаги более детально.

Разрешение перегрузки функции не применяется, если все операнды имеют встроенные типы. В таком случае гарантированно употребляется встроенный оператор. (Использование операторов с операндами встроенных типов описано в главе 4.) Например:

```
class SmallInt {
public:
   SmallInt( int );
};

SmallInt operator+ ( const SmallInt &, const SmallInt & );
void func() {
   int i1, i2;
   int i3 = i1 + i2;

}
```

Поскольку операнды `i1` и `i2` имеют тип `int`, а не тип класса, то при сложении используется встроенный оператор +. Перегруженный `operator+(const SmallInt &, const SmallInt &)` игнорируется, хотя операнды можно привести к типу `SmallInt` с помощью определенного пользователем преобразования в виде конструктора `SmallInt(int)`. Описанный ниже процесс разрешения перегрузки в таких ситуациях не применяется.

Кроме того, разрешение перегрузки для операторов употребляется только в случае

```
void func() {
   SmallInt si(98);
   int iobj = 65;
   int res = si + iobj;  // использован операторный синтаксис
}
```

использования операторного синтаксиса:

Если вместо этого использовать синтаксис вызова функции:

```
int res = operator+( si, iobj );  // синтаксис вызова функции
```

то применяется процедура разрешения перегрузки для функций в пространстве имен (см.

```
// синтаксис вызова функции-члена
```

раздел 15.10). Если же использован синтаксис вызова функции-члена:

```
int res = si.operator+( iobj );
```

то работает соответствующая процедура для функций-членов (см. раздел 15.11).

15.12.1. Операторные функции-кандидаты

Операторная функция является кандидатом, если она имеет то же имя, что и вызванная.

```
SmallInt si(98);
int iobj = 65;
```

При использовании следующего оператора сложения

```
int res = si + iobj;
```

операторной функцией-кандидатом является operator+. Какие объявления operator+ принимаются во внимание?

Потенциально в случае применения операторного синтаксиса с операндами, имеющими тип класса, строится пять множеств кандидатов. Первые три – те же, что и при вызове обычных функций с аргументами типа класса:

- множество операторов, видимых в точке вызова. Объявления функции operator+(), видимые в точке использования оператора, являются кандидатами. Например, operator+(), объявленный в глобальной области видимости, –

```
SmallInt operator+ ( const SmallInt &, const SmallInt & );

int main() {
   SmallInt si(98);
   int iobj = 65;
   int res = si + iobj;  // ::operator+() - функция-кандидат
```

кандидат в случае применения operator+() внутри main():

```
}
```

- множество операторов, объявленных в пространстве имен, в котором определен тип операнда. Если операнд имеет тип класса и этот тип объявлен в пользовательском пространстве имен, то операторные функции, объявленные в том же пространстве и имеющие то же имя, что и использованный оператор,

```
namespace NS {
   class SmallInt { /* ... */ };
   SmallInt operator+ ( const SmallInt&, double );
}

int main() {
   // si имеет тип SmallInt:
   // этот класс объявлен в пространстве имен NS
   NS::SmallInt si(15);

   // NS::operator+() - функция-кандидат
   int res = si + 566;
   return 0;
```

считаются кандидатами:

```
}
```

Операнд si имеет тип класса SmallInt, объявленного в пространстве имен NS. Поэтому перегруженный operator+(const SmallInt, double), объявленный в том же пространстве, добавляется к множеству кандидатов;

- множество операторов, объявленных друзьями классов, к которым принадлежат операнды. Если операнд принадлежит к типу класса и в определении этого класса есть одноименные применяемому оператору функции-друзья, то они добавляются к множеству кандидатов:

```
namespace NS {
   class SmallInt {
      friend SmallInt operator+( const SmallInt&, int )
                                { /* ... */ }
   };
}
int main() {
   NS::SmallInt si(15);

   // функция-друг operator+() - кандидат
   int res = si + 566;
   return 0;
}
```

Операнд `si` имеет тип `SmallInt`. Операторная функция `operator+(const SmallInt&, int)`, являющаяся другом этого класса, – член пространства имен `NS`, хотя непосредственно в этом пространстве она не объявлена. При обычном поиске в `NS` эта операторная функция не будет найдена. Однако при использовании `operator+()` с аргументом типа `SmallInt` функции-друзья, объявленные в области видимости этого класса, включаются в рассмотрение и добавляются к множеству кандидатов.

Эти три множества операторных функций-кандидатов формируются точно так же, как и для вызовов обычных функций с аргументами типа класса. Однако при использовании операторного синтаксиса строятся еще два множества:

- множество операторов-членов, объявленных в классе левого операнда. Если такой операнд оператора `operator+()` имеет тип класса, то в множество функций-кандидатов включаются объявления `operator+()`, являющиеся членами этого

```
class myFloat {
   myFloat( double );
};
class SmallInt {
public:
   SmallInt( int );
   SmallInt operator+ ( const myFloat & );
};

int main() {
   SmallInt si(15);

   int res = si + 5.66;  // оператор-член operator+() - кандидат
```

   класса:

```
}
```

Оператор-член `SmallInt::operator+(const myFloat &)`, определенный в `SmallInt`, включается в множество функций-кандидатов для разрешения вызова `operator+()` в `main()`;

- множество встроенных операторов. Учитывая типы, которые можно использовать со встроенным `operator+()`, кандидатами являются также:

```
int operator+( int, int );
double operator+( double, double );
T* operator+( T*, I );

T* operator+( I, T* );
```

Первое объявление относится к встроенному оператору для сложения двух значений целых типов, второе – к оператору для сложения значений типов с плавающей точкой. Третье и четвертое соответствуют встроенному оператору сложения указательных типов, который используется для прибавления целого числа к указателю. Два последних объявления представлены в символическом виде и описывают целое семейство встроенных операторов, которые могут быть выбраны компилятором на роль кандидатов при обработке операций сложения.

Любое из первых четырех множеств может оказаться пустым. Например, если среди членов класса `SmallInt` нет функции с именем `operator+()`, то четвертое множество будет пусто.

Все множество операторных функций-кандидатов является объединением пяти

```
namespace NS {
   class myFloat {
      myFloat( double );
   };
   class SmallInt {
      friend SmallInt operator+( const SmallInt &, int ) { /* ... */ }
   public:
      SmallInt( int );
      operator int();
      SmallInt operator+ ( const myFloat & );
      // ...
   };
   SmallInt operator+ ( const SmallInt &, double );
}

int main() {
   // тип si - class SmallInt:
   // Этот класс объявлен в пространстве имен NS
   NS::SmallInt si(15);

   int res = si + 5.66;  // какой operator+()?
   return 0;
}
```

подмножеств, описанных выше:

В эти пять множеств входят семь операторных функций-кандидатов на роль `operator+()` в `main()`:

- первое множество пусто. В глобальной области видимости, а именно в ней употреблен `operator+()` в функции `main()`, нет объявлений перегруженного оператора `operator+()`;
- второе множество содержит операторы, объявленные в пространстве имен `NS`, где определен класс `SmallInt`. В этом пространстве имеется один оператор:

```
NS::SmallInt NS::operator+( const SmallInt &, double );
```

- третье множество содержит операторы, объявленные друзьями класса `SmallInt`. Сюда входит

```
NS::SmallInt NS::operator+( const SmallInt &, int );
```

- четвертое множество содержит операторы, объявленные членами `SmallInt`. Такой тоже есть:

```
NS::SmallInt NS::SmallInt::operator+( const myFloat & );
```

```
int operator+( int, int );
double operator+( double, double );
T* operator+( T*, I );
```

- пятое множество содержит встроенные бинарные операторы:

```
T* operator+( I, T* );
```

Да, формирование множества кандидатов для разрешения оператора, использованного с применением операторного синтаксиса, утомительно. Но после того как оно построено, устоявшие функции и наилучшая из них находятся, как и прежде, путем анализа преобразований, применимых к операндам отобранных кандидатов.

15.12.2. Устоявшие функции

Множество устоявших операторных функций формируется из множества кандидатов путем отбора лишь тех операторов, которые могут быть вызваны с заданными операндами. Например, какие из семи найденных выше кандидатов устоят? Оператор

```
NS::SmallInt si(15);
```

использован в следующем контексте:

```
si + 5.66;
```

Левый операнд имеет тип `SmallInt`, а правый – `double`.

Первый кандидат является устоявшей функцией для данного использования `operator+()`:

```
NS::SmallInt NS::operator+( const SmallInt &, double );
```

Левый операнд типа `SmallInt` в качестве инициализатора точно соответствует формальному параметру-ссылке этого перегруженного оператора. Правый, имеющий тип `double`, также точно соответствует второму формальному параметру.

Следующая функция-кандидат также устоит:

```
NS::SmallInt NS::operator+( const SmallInt &, int );
```

Левый операнд `si` типа `SmallInt` в качестве инициализатора точно соответствует формальному параметру-ссылке перегруженного оператора. Правый имеет тип `int` и

может быть приведен к типу второго формального параметра с помощью стандартного преобразования.

Устоит и третья функция-кандидат:

```
NS::SmallInt NS::SmallInt::operator+( const myFloat & );
```

Левый операнд `si` имеет тип `SmallInt`, т.е. тип того класса, членом которого является перегруженный оператор. Правый имеет тип `int` и приводится к типу класса `myFloat` с помощью определенного пользователем преобразования в виде конструктора `myFloat(double)`.

```
int operator+( int, int );
```

Четвертой и пятой устоявшими функциями являются встроенные операторы:

```
double operator+( double, double );
```

Класс `SmallInt` содержит конвертер, который может привести значение типа `SmallInt` к типу `int`. Этот конвертер используется вместе с первым встроенным оператором для преобразования левого операнда в тип `int`. Второй операнд типа `double` трансформируется в тип `int` с помощью стандартного преобразования. Что касается второго встроенного оператора, то конвертер приводит левый операнд от типа `SmallInt` к типу `int`, после чего результат стандартно преобразуется в `double`. Второй же операнд типа `double` точно соответствует второму параметру.

Лучшей из этих пяти устоявших функций является первая, `operator+()`, объявленная в пространстве имен `NS`:

```
NS::SmallInt NS::operator+( const SmallInt &, double );
```

Оба ее операнда точно соответствуют параметрам.

15.12.3. Неоднозначность

Наличие в одном и том же классе конвертеров, выполняющих неявные преобразования во встроенные типы, и перегруженных операторов может приводить к неоднозначности при выборе между ними. Например, есть следующее определение класса `String` с функцией

```
class String {
   // ...
public:
   String( const char * = 0 );
   bool operator== ( const String & ) const;
   // нет оператора operator== ( const char * )
```

сравнения:

```
};
```

и такое использование оператора `operator==`:

```
String flower( "tulip" );
void foo( const char *pf ) {
   // вызывается перегруженный оператор String::operator==()
   if ( flower == pf )
      cout << pf << " is a flower!\en";
      // ...
}
```

Тогда при сравнении

```
flower == pf
```

вызывается оператор равенства класса `String`:

```
String::operator==( const String & ) const;
```

Для трансформации правого операнда `pf` из типа `const char*` в тип `String` параметра `operator==()` применяется определенное пользователем преобразование, которое вызывает конструктор:

```
String( const char * )
```

```
class String {
   // ...
public:
   String( const char * = 0 );
   bool operator== ( const String & ) const;
   operator const char*();  // новый конвертер
};
```

Если добавить в определение класса `String` конвертер в тип `const char*`:

```
// проверка на равенство больше не компилируется!
```

то показанное использование `operator==()` становится неоднозначным:

```
if (flower == pf)
```

Из-за добавления конвертера `operator const char*()` встроенный оператор сравнения

```
bool operator==( const char *, const char * )
```

тоже считается устоявшей функцией. С его помощью левый операнд `flower` типа `String` может быть преобразован в тип `const char *`.

Теперь для использования `operator==()` в `foo()` есть две устоявших операторных функции. Первая из них

```
String::operator==( const String & ) const;
```

требует применения определенного пользователем преобразования правого операнда pf из типа const char* в тип String. Вторая

```
bool operator==( const char *, const char * )
```

требует применения пользовательского преобразования левого операнда flower из типа String в тип const char*.

Таким образом, первая устоявшая функция лучше для левого операнда, а вторая – для правого. Поскольку наилучшей функции не существует, то вызов помечается компилятором как неоднозначный.

При проектировании интерфейса класса, включающего объявление перегруженных операторов, конструкторов и конвертеров, следует быть весьма аккуратным. Определенные пользователем преобразования применяются компилятором неявно. Это может привести к тому, что встроенные операторы окажутся устоявшими при разрешении перегрузки для операторов с операндами типа класса.

Упражнение 15.17

Назовите пять множеств функций-кандидатов, рассматриваемых при разрешении перегрузки оператора с операндами типа класса.

Упражнение 15.18

Какой из операторов operator+() будет выбран в качестве наилучшего из устоявших для оператора сложения в main()? Перечислите все функции-кандидаты, все устоявшие функции и преобразования типов, которые надо применить к аргументам для каждой

```
namespace NS {
   class complex {
      complex( double );
      // ...
   };
   class LongDouble {
      friend LongDouble operator+( LongDouble &, int ) { /* ... */ }
   public:
      LongDouble( int );
      operator double();
      LongDouble operator+( const complex & );
      // ...
   };
   LongDouble operator+( const LongDouble &, double );
}

int main() {
   NS::LongDouble ld(16.08);

   double res = ld + 15.05;   // какой operator+?
   return 0;
}
```

устоявшей функции.

16. Шаблоны классов

В этой главе описывается, как определять и использовать шаблоны классов. Шаблон – это предписание для создания класса, в котором один или несколько типов либо значений параметризованы. Начинающий программист может использовать шаблоны, не понимая механизма, стоящего за их определениями и конкретизациями. Фактически на протяжении всей этой книги мы пользовались шаблонами классов, которые определены в стандартной библиотеке C++ (например, vector, list и т.д.), и при этом не нуждались в детальном объяснении механизма их работы. Только профессиональные программисты определяют собственные шаблоны классов и пользуются описанными в данной главе средствами. Поэтому этот материал следует рассматривать как введение в более сложные аспекты C++.

Глава 16 содержит вводные и продвинутые разделы. Во вводных разделах показано, как определяются шаблоны классов, иллюстрируются простые способы применения и обсуждается механизм их конкретизации. Мы расскажем, как можно задавать в шаблонах разные виды членов: функции-члены, статические данные-члены и вложенные типы. В продвинутых разделах представлен материал, необходимый для написания приложений промышленного уровня. Сначала мы рассмотрим, как компилятор конкретизирует шаблоны и какие требования в связи с этим предъявляются к организации нашей программы. Затем покажем, как определять специализации и частичные специализации для шаблона класса и для его члена. Далее мы остановимся на двух вопросах, представляющих интерес для проектировщиков: как разрешаются имена в определениях шаблона класса и как можно определять шаблоны в пространствах имен. Завершается эта глава примером определения и использования шаблона класса.

16.1. Определение шаблона класса

Предположим, что нам нужно определить класс, поддерживающий механизм очереди. Очередь – это структура данных для хранения коллекции объектов; они помещаются в конец очереди, а извлекаются из ее начала. Поведение очереди описывают аббревиатурой FIFO – “первым пришел, первым ушел”. (Определенный в стандартной библиотеке C++ тип, реализующий очередь, упоминался в разделе 6.17. В этой главе мы создадим упрощенный тип для знакомства с шаблонами классов.)

Необходимо, чтобы наш класс Queue поддерживал следующие операции:

- добавить элемент в конец очереди:

```
void add( item );
```

- удалить элемент из начала очереди:

```
item remove();
```

- определить, пуста ли очередь:

```
bool is_empty();
```

- определить, заполнена ли очередь:

```
bool is_full();
```

```
class Queue {
public:
   Queue();
   ~Queue();

   Type& remove();
   void add( const Type & );
   bool is_empty();
   bool is_full();
private:
   // ...
```

Определение Queue могло бы выглядеть так:

```
};
```

Вопрос в том, какой тип использовать вместо Type? Предположим, что мы решили реализовать класс Queue, заменив Type на int. Тогда Queue может управлять коллекциями объектов типа int. Если бы понадобилось поместить в очередь объект другого типа, то его пришлось бы преобразовать в тип int, если же это невозможно,

```
Queue qObj;
string str( "vivisection" );

qObj.add( 3.14159 );  // правильно: в очередь помещен объект 3
```

компилятор выдаст сообщение об ошибке:

```
qObj.add( str );      // ошибка: нет преобразования из string в int
```

Поскольку любой объект в коллекции имеет тип int, то язык C++ гарантирует, что в очередь можно поместить либо значение типа int, либо значение, преобразуемое в такой тип. Это подходит, если предстоит работа с очередями объектов только типа int. Если же класс Queue должен поддерживать также коллекции объектов типа double, char, комплексные числа или строки, подобная реализация оказывается слишком ограничительной.

Конечно, эту проблему можно решить, создав копию класса Queue для работы с типом double, затем для работы с комплексными числами, затем со строками и т.д. А поскольку имена классов перегружать нельзя, каждой реализации придется дать уникальное имя: IntQueue, DoubleQueue, ComplexQueue, StringQueue. При необходимости работать с другим классом придется снова копировать, модифицировать и переименовывать.

Такой метод дублирования кода крайне неудобен. Создание различных уникальных имен для Queue представляет лексическую сложность. Имеются и трудности администрирования: любое изменение общего алгоритма придется вносить в каждую реализацию класса. В общем случае процесс ручной генерации копий для индивидуальных типов никогда не кончается и очень сложен с точки зрения сопровождения.

К счастью, механизм шаблонов C++ позволяет автоматически генерировать такие типы. Шаблон класса можно использовать при создании Queue для очереди объектов любого

```
template <class Type>
class Queue {
public:
   Queue();
   ~Queue();

   Type& remove();
   void add( const Type & );
   bool is_empty();
   bool is_full();
private:
   // ...
```

типа. Определение шаблона этого класса могло бы выглядеть следующим образом:

```
};
```

Чтобы создать классы Queue, способные хранить целые числа, комплексные числа и

```
Queue<int> qi;
Queue< complex<double> > qc;
```

строки, программисту достаточно написать:

```
Queue<string> qs;
```

Реализация Queue представлена в следующих разделах с целью иллюстрации определения и применения шаблонов классов. В реализации используются две абстракции шаблона:

- сам шаблон класса Queue предоставляет описанный выше открытый интерфейс и пару членов: front и back. Очередь реализуется с помощью связанного списка;
- шаблон класса QueueItem представляет один узел связанного списка Queue. Каждый помещаемый в очередь элемент сохраняется в объекте QueueItem, который содержит два члена: value и next. Тип value будет различным в каждом экземпляре класса Queue, а next – это всегда указатель на следующий объект QueueItem в очереди.

Прежде чем приступать к детальному изучению реализации этих шаблонов, рассмотрим,

```
template <class T>
```

как они объявляются и определяются. Вот объявление шаблона класса QueueItem:

```
   class QueueItem;
```

Как объявление, так и определение шаблона всегда начинаются с ключевого слова template. За ним следует заключенный в угловые скобки *список параметров шаблона*, разделенных запятыми. Список не бывает пустым. В нем могут быть *параметры-типы*, представляющие некоторый тип, и *параметры-константы*, представляющие некоторое константное выражение.

Параметр-тип шаблона состоит из ключевого слова `class` или `typename` (в списке параметров они эквивалентны), за которым следует идентификатор. (Ключевое слово `typename` не поддерживается компиляторами, написанными до принятия стандарта C++. В разделе 10.1 подробно объяснялось, зачем это слово было добавлено в язык.) Оба ключевых слова обозначают, что последующее имя параметра относится к встроенному или определенному пользователем типу. Например, в приведенном выше определении шаблона `QueueItem` имеется один параметр-тип `T`. Допустимым фактическим аргументом для `T` является любой встроенный или определенный пользователем тип, такой, как `int`, `double`, `char*`, `complex` или `string`.

```
template <class T1, class T2, class T3>
```

У шаблона класса может быть несколько параметров-типов:

```
class Container;
```

Однако ключевое слово `class` или `typename` должно предшествовать каждому.

```
// ошибка: должно быть <typename T, class U> или
//                     <typename T, typename U>
template <typename T, U>
```

Следующее объявление ошибочно:

```
class collection;
```

Объявленный параметр-тип служит спецификатором типа в оставшейся части определения шаблона и употребляется точно так же, как любой встроенный или определенный пользователем тип в обычном определении класса. Например, параметр-тип можно использовать для объявления данных и функций-членов, членов вложенных классов и т.д.

Не являющийся типом параметр шаблона представляет собой обычное объявление. Он показывает, что следующее за ним имя – это потенциальное значение, употребляемое в определении шаблона в качестве константы. Так, шаблон класса `Buffer` может иметь параметр-тип, представляющий типы элементов, хранящихся в буфере, и параметр-

```
template <class Type, int size>
```

константу, содержащий его размер:

```
class Buffer;
```

За списком параметров шаблона следует определение или объявление класса. Шаблон определяется так же, как обычный класс, но с указанием параметров:

```
template <class Type>
class QueueItem {
public:
   // ...
private:
   // Type представляет тип члена
   Type item;
   QueueItem *next;
};
```

В этом примере Type используется для обозначения типа члена item. По ходу выполнения программы вместо Type могут быть подставлены различные встроенные или определенные пользователем типы. Такой процесс подстановки называется *конкретизацией* шаблона.

Имя параметра шаблона можно употреблять после его объявления и до конца объявления или определения шаблона. Если в глобальной области видимости объявлена переменная с таким же именем, как у параметра шаблона, то это имя будет скрыто. В следующем

```
typedef double Type;

template <class Type>
class QueueItem {
public:
   // ...
private:
   // тип Item - не double
   Type item;
   QueueItem *next;
};
```

примере тип item равен не double, а типу параметра:

```
template <class Type>
class QueueItem {
public:
   // ...
private:
   // ошибка: член не может иметь то же имя, что и
   //         параметр шаблона Type
   typedef double Type;
   Type item;
   QueueItem *next;
};
```

Член класса внутри определения шаблона не может быть одноименным его параметру:

Имя параметра шаблона может встречаться в списке только один раз. Поэтому следующее объявление компилятор помечает как ошибку:

```
// ошибка: неправильное использование имени параметра шаблона Type
template <class Type, class Type>
   class container;
```

Такое имя разрешается повторно использовать в объявлениях или определениях других

```
// правильно: повторное использование имени Type в разных шаблонах
template <class Type>
   class QueueItem;

template <class Type>
```

шаблонов:

```
   class Queue;
```

Имена параметров в опережающем объявлении и последующем определении одного и того же шаблона не обязаны совпадать. Например, все эти объявления QueueItem

```
// все три объявления QueueItem
// относятся к одному и тому же шаблону класса

// объявления шаблона
template <class T> class QueueItem;
template <class U> class QueueItem;

// фактическое определение шаблона
template <class Type>
```

относятся к одному шаблону класса:

```
   class QueueItem { ... };
```

У параметров могут быть аргументы по умолчанию (это справедливо как для параметров-типов, так и для параметров-констант) – тип или значение, которые используются в том случае, когда при конкретизации шаблона фактический аргумент не указан. В качестве такого аргумента следует выбирать тип или значение, подходящее для большинства конкретизаций. Например, если при конкретизации шаблона класса Buffer не указан

```
template <class Type, size = 1024>
```

размер буфера, то по умолчанию принимается 1024:

```
   class Buffer;
```

В последующих объявлениях шаблона могут быть заданы дополнительные аргументы по умолчанию. Как и в объявлениях функций, если для некоторого параметра задан такой аргумент, то он должен быть задан и для всех параметров, расположенных в списке

```
template <class Type, size = 1024>
```

правее (даже в другом объявлении того же шаблона):

```
   class Buffer;
```

```
// правильно: рассматриваются аргументы по умолчанию из обоих объявлений
template <class Type=string, int size>

   class Buffer;
```

(Отметим, что аргументы по умолчанию для параметров шаблонов не поддерживаются в компиляторах, реализованных до принятия стандарта C++. Чтобы примеры из этой книги, в частности из главы 12, компилировались большинством современных компиляторов, мы не использовали такие аргументы.)

Внутри определения шаблона его имя можно применять как спецификатор типа всюду, где допустимо употребление имени обычного класса. Вот более полная версия

```
template <class Type>
class QueueItem {
public:
   QueueItem( const Type & );
private:
   Type item;
   QueueItem *next;
};
```

определения шаблона `QueueItem`:

Обратите внимание, что каждое появление имени `QueueItem` в определении шаблона – это сокращенная запись для

```
QueueItem<Type>
```

Такую сокращенную нотацию можно употреблять только внутри определения `QueueItem` (и, как мы покажем в следующих разделах, в определениях его членов, которые находятся вне определения шаблона класса). Если `QueueItem` применяется как спецификатор типа в определении какого-либо другого шаблона, то необходимо задавать полный список параметров. В следующем примере шаблон класса используется в определении шаблона функции `display`. Здесь за именем шаблона класса `QueueItem` должны идти параметры,

```
template <class Type>
void display( QueueItem<Type> &qi )
{
   QueueItem<Type> *pqi = &qi;
   // ...
}
```

т.е. `QueueItem<Type>`.

16.1.1. Определения шаблонов классов Queue и QueueItem

Ниже представлено определение шаблона класса `Queue`. Оно помещено в заголовочный файл `Queue.h` вместе с определением шаблона `QueueItem`:

```
#ifndef QUEUE_H
#define QUEUE_H

// объявление QueueItem
template <class T> class QueueItem;

template <class Type>
class Queue {
public:
   Queue() : front( 0 ), back ( 0 ) { }
   ~Queue();

   Type& remove();
   void add( const Type & );
   bool is_empty() const {
      return front == 0;
   }
private:
   QueueItem<Type> *front;
   QueueItem<Type> *back;
};

#endif
```

При использовании имени Queue внутри определения шаблона класса Queue список параметров <Type> можно опускать. Однако пропуск списка параметров шаблона QueueItem в определении шаблона Queue недопустим. Так, объявление члена front

```
template <class Type>
class Queue {
public:
   // ...
private:
   // ошибка: список параметров для QueueItem неизвестен
   QueueItem<Type> *front;
}
```

является ошибкой:

Упражнение 16.1

```
(a) template <class Type>
       class Container1;

    template <class Type, int size>
```

Найдите ошибочные объявления (или пары объявлений) шаблонов классов:

```
(b) template <class T, U, class V>
       class Container1;

       class Container2;
```

```
(c) template <class C1, typename C2>


(d) template <typename myT, class myT>




(e) template <class Type, int *pi>
       class Container3 {};

       class Container4 {};


(f) template <class Type, int val = 0>
       class Container6;
    template <class T = complex<double>, int v>
       class Container5;

       class Container6;
```

Упражнение 16.2

```
template <class elemenType>
class ListItem;

template <class elemType>
class List {
public:
   List<elemType>()
       : _at_front( 0 ), _at_end( 0 ), _current( 0 ), _size( 0 )
       {}
   List<elemType>( const List<elemType> & );
   List<elemType>& operator=( const List<elemType> & );

   ~List();

   void insert( ListItem *ptr, elemType value );
   int  remove( elemType value );

   ListItem *find( elemType value );

   void display( ostream &os = cout );
   int size() { return _size; }
private:
   ListItem *_at_front;
   ListItem *_at_end;
   ListItem *_current;
   int _size
```

Следующее определение шаблона `List` некорректно. Как исправить ошибку?

```
};
```

16.2. Конкретизация шаблона класса

В определении шаблона указывается, как следует строить индивидуальные классы, если заданы один или более фактических типов или значений. По шаблону Queue автоматически генерируются экземпляры классов Queue с разными типами элементов. Например, если написать:

```
Queue<int> qi;
```

то из обобщенного определения шаблона автоматически создается класс Queue для объектов типа int.

Генерация конкретного класса из обобщенного определения шаблона называется *конкретизацией шаблона*. При такой конкретизации Queue для объектов типа int каждое вхождение параметра Type в определении шаблона заменяется на int, так что

```
template <class int>
class Queue {
public:
   Queue() : front( 0 ), back ( 0 ) { }
   ~Queue();

   int& remove();
   void add( const int & );
   bool is_empty() const {
      return front == 0;
   }
private:
   QueueItem<int> *front;
   QueueItem<int> *back;
```

определение класса Queue принимает вид:

```
};
```

Чтобы создать класс Queue для объектов типа string, надо написать:

```
Queue<string> qs;
```

При этом каждое вхождение Type в определении шаблона будет заменено на string. Объекты qi и qs являются объектами автоматически созданных классов.

Каждый конкретизированный по одному и тому же шаблону экземпляр класса совершенно не зависит от всех остальных. Так, у Queue для типа int нет никаких прав доступа к неоткрытым членам того же класса для типа string.

Конкретизированный экземпляр шаблона будет иметь соответственно имя Queue<int> или Queue<string>. Части <int> и <string>, следующие за именем Queue, называются фактическими аргументами шаблона. Они должны быть заключены в угловые скобки и отделяться друг от друга запятыми. В имени конкретизируемого шаблона аргументы всегда должны задаваться явно. В отличие от аргументов шаблона функции, аргументы шаблона класса никогда не выводятся из контекста:

```
Queue qs;  // ошибка: как конкретизируется шаблон?
```

Конкретизированный шаблон класса Queue можно использовать в программе всюду, где

```
// типы возвращаемого значения и обоих параметров конкретизированы из
// шаблона класса Queue
extern Queue< complex<double> >
   foo( Queue< complex<double> > &, Queue< complex<double> > & );

// указатель на функцию-член класса, конкретизированного из шаблона Queue
bool (Queue<double>::*pmf)() = 0;

// явное приведение 0 к указателю на экземпляр Queue
```

допустимо употребление типа обычного класса:

```
Queue<char*> *pqc = static_cast< Queue<char*>* > ( 0 );
```

Объекты типа класса, конкретизированного по шаблону Queue, объявляются и

```
extern Queue<double> eqd;
Queue<int> *pqi = new Queue<int>;
Queue<int> aqi[1024];

int main() {
   int ix;
   if ( ! pqi->is_empty() )
      ix = pqi->remove();
   // ...
   for ( ix = 0; ix < 1024; ++ix )
       eqd[ ix ].add( ix );
   // ...
```

используются так же, как объекты обычных классов:

```
}
```

В объявлении и определении шаблона можно ссылаться как на сам шаблон, так и на

```
// объявление шаблона функции
template <class Type>
void bar( Queue<Type> &,   // ссылается на обобщенный шаблон
          Queue<double> &  // ссылается на конкретизированный шаблон
```

конкретизированный по нему класс:

```
)
```

Однако вне такого определения употребляются только конкретизированные экземпляры. Например, в теле обычной функции всегда надо задавать фактические аргументы

```
void foo( Queue<int> &qi )
{
   Queue<int> *pq = &qi;
   // ...
```

шаблона Queue:

```
}
```

Шаблон класса конкретизируется только тогда, когда имя полученного экземпляра употребляется в контексте, где требуется определение шаблона. Не всегда определение класса должно быть известно. Например, перед объявлением указателей и ссылок на

```
class Matrix;
Matrix *pm;   // правильно: определение класса Matrix знать необязательно
```

класс его знать необязательно:

```
void inverse( Matrix & );  // тоже правильно
```

Поэтому объявление указателей и ссылок на конкретизированный шаблон класса не приводит к его конкретизации. (Отметим, что в некоторых компиляторах, написанных до принятия стандарта C++, шаблон конкретизируется при первом упоминании имени конкретизированного класса в тексте программы.) Так, в функции `foo()` объявляются

```
// Queue<int> не конкретизируется при таком использовании в foo()
void foo( Queue<int> &qi )
{
   Queue<int> *pqi = &qi;
   // ...
```

указатель и ссылка на `Queue<int>`, но это не вызывает конкретизации шаблона `Queue`:

```
}
```

Определение класса необходимо знать, когда определяется объект этого типа. В следующем примере определение `obj1` ошибочно: чтобы выделить для него память,

```
class Matrix;
Matrix obj1;   // ошибка: класс Matrix не определен
class Matrix { ... };
```

компилятору необходимо знать размер класса `Matrix`:

```
Matrix obj2;  // правильно
```

Таким образом, конкретизация происходит тогда, когда определяется объект класса, конкретизированного по этому шаблону. В следующем примере определение объекта `qi` приводит к конкретизации шаблона `Queue<int>`:

```
Queue<int> qi;   // конкретизируется Queue<int>
```

Определение `Queue<int>` становится известно компилятору именно в этой точке, которая называется *точкой конкретизации* данного класса.

Если имеется указатель или ссылка на конкретизированный шаблон, то конкретизация также производится в момент обращения к объекту, на который они ссылаются. В определенной выше функции `foo()` класс `Queue<int>` конкретизируется в следующих случаях: когда разыменовывается указатель `pqi`, когда ссылка `qi` используется для

получения значения именуемого объекта и когда pqi или qi употребляются для доступа к

```
void foo( Queue<int> &qi )
{
   Queue<int> *pqi = &qi;

   // Queue<int> конкретизируется в результате вызова функции-члена
   pqi->add( 255 );
   // ...
```

членам или функциям-членам этого класса:

```
}
```

Определение Queue<int> становится известным компилятору еще до вызова функции-члена add() из foo().

Напомним, что в определении шаблона класса Queue есть также ссылка на шаблон

```
template <class Type>
class Queue {
public:
   // ...
private:
   QueueItem<Type> *front;
   QueueItem<Type> *back;
```

QueueItem:

```
};
```

При конкретизации Queue типом int члены front и back становятся указателями на QueueItem<int>. Следовательно, конкретизированный экземпляр Queue<int> ссылается на экземпляр QueueItem, конкретизированный типом int. Но поскольку соответствующие члены являются указателями, то QueueItem<int> конкретизируется лишь в момент их разыменования в функциях-членах класса Queue<int>.

Наш класс QueueItem служит вспомогательным средством для реализации класса Queue и не будет непосредственно употребляться в вызывающей программе. Поэтому пользовательская программа способна манипулировать только объектами Queue. Конкретизация шаблона QueueItem происходит лишь в момент конкретизации шаблона класса Queue или его членов. (В следующих разделах мы рассмотрим конкретизации членов шаблона класса.)

В зависимости от типов, которыми может конкретизироваться шаблон, при его определении надо учитывать некоторые нюансы. Почему, например, следующее определение конструктора класса QueueItem не подходит для конкретизации общего

```
template <class Type>
class QueueItem {
public:
   QueueItem( Type );   // неудачное проектное решение
   // ...
```

вида?

```
};
```

В данном определении аргумент передается по значению. Это допустимо, если QueueItem конкретизируется встроенным типом (например, QueueItem<int>). Но если такая конкретизация производится для объемного типа (скажем, Matrix), то накладные расходы, вызванные неправильным выбором на этапе проектирования, становятся неприемлемыми. (В разделе 7.3 обсуждались вопросы производительности, связанные с передачей параметров по значению и по ссылке.) Поэтому аргумент конструктора объявляется как ссылка на константный тип:

```
QueueItem( const Type & );
```

Следующее определение приемлемо, если у типа, для которого конкретизируется

```
template <class Type>
class QueueItem {
   // ...
public:
   // потенциально неэффективно
   QueueItem( const Type &t ) {
      item = t; next = 0;
   }
```

QueueItem, нет ассоциированного конструктора:

```
};
```

Если аргументом шаблона является тип класса с конструктором (например, string), то item инициализируется дважды! Конструктор по умолчанию string вызывается для инициализации item перед выполнением тела конструктора QueueItem. Затем для созданного объекта item производится почленное присваивание. Избежать такого можно с помощью явной инициализации item в списке инициализации членов внутри

```
template <class Type>
class QueueItem {
   // ...
public:
   // item инициализируется в списке инициализации членов конструктора
   QueueItem( const Type &t )
            : item(t) { next = 0; }
```

определения конструктора QueueItem:

```
};
```

(Списки инициализации членов и основания для их применения обсуждались в разделе 14.5.)

16.2.1. Аргументы шаблона для параметров-констант

Параметр шаблона класса может и не быть типом. На аргументы, подставляемые вместо таких параметров, накладываются некоторые ограничения. В следующем примере мы изменяем определение класса Screen (см. главу 13) на шаблон, параметризованный высотой и шириной:

```
template <int hi, int wid>
class Screen {
public:
   Screen() : _height( hi ), _width( wid ), _cursor ( 0 ),
              _screen( hi * wid, '#' )
            { }
   // ...
private:
   string            _screen;
   string::size_type _cursor;
   short             _height;
   short             _width;
};

typedef Screen<24,80> termScreen;
termScreen hp2621;

Screen<8,24> ancientScreen;
```

Выражение, с которым связан параметр, не являющийся типом, должно быть константным, т.е. вычисляемым во время компиляции. В примере выше `typedef termScreen` ссылается на экземпляр шаблона `Screen<24,80>`, где аргумент шаблона для `hi` равен `24`, а для `wid` – `80`. В обоих случаях аргумент – это константное выражение.

Однако для шаблона `BufPtr` конкретизация приводит к ошибке, так как значение указателя, получающееся при вызове оператора `new()`, становится известно только во

```
template <int *ptr> class BufPtr { ... };

// ошибка: аргумент шаблона нельзя вычислить во время компиляции
```

время выполнения:

```
BufPtr< new int[24] > bp;
```

Не является константным выражением и значение неконстантного объекта. Его нельзя использовать в качестве аргумента для параметра-константы шаблона. Однако адрес любого объекта в области видимости пространства имен, в отличие от адреса локального объекта, является константным выражением (даже если спецификатор `const` отсутствует), поэтому его можно применять в качестве аргумента для параметра-

```
template <int size> Buf { ... };
template <int *ptr> class BufPtr { ... };

int size_val = 1024;
const int c_size_val = 1024;

Buf< 1024 > buf0;   // правильно
Buf< c_size_val > buf1; // правильно
Buf< sizeof(size_val) > buf2; // правильно: sizeof(int)
BufPtr< &size_val > bp0; // правильно

// ошибка: нельзя вычислить во время компиляции
```

константы. Константным выражением будет и значение оператора `sizeof`:

```
Buf< size_val > buf3;
```

Вот еще один пример, иллюстрирующий использование параметра-константы для представления константного значения в определении шаблона, а также применение его

```
template < class Type, int size >
class FixedArray {
public:
   FixedArray( Type *ar ) : count( size )
   {
      for ( int ix = 0; ix < size; ++ix )
          array[ ix ] = ar[ ix ];
   }
private:
   Type array[ size ];
   int count;
};

int ia[4] = { 0, 1, 2, 3 };
```

аргумента для задания значения этого параметра:

```
FixedArray< int, sizeof( is ) / sizeof( int ) > iA{ ia );
```

Выражения с одинаковыми значениями считаются эквивалентными аргументами для параметров-констант шаблона. Так, все три экземпляра Screen ссылаются на один и тот

```
const int width = 24;
const int height = 80;

// все это Screen< 24, 80 >
Screen< 2*12, 40*2 > scr0;
Screen< 6+6+6+6, 20*2 + 40 > scr1;
```

же конкретизированный из шаблона класс Screen<24,80>:

```
Screen< width, height > scr2;
```

Между типом аргумента шаблона и типом параметра-константы допустимы некоторые преобразования. Их множество является подмножеством преобразований, допустимых для аргументов функции:

- трансформации l-значений, включающие преобразование l-значения в r-

```
template <int *ptr> class BufPtr { ... };

int array[10];
```

значение, массива в указатель и функции в указатель:

```
BufPtr< array > bpObj;  // преобразование массива в указатель
```

```
template <const int *ptr> class Ptr { ... };

int iObj;
```

- преобразования квалификаторов:

```
Ptr< &iObj > pObj;   // преобразование из int* в const int*
```

```
template <int hi, int wid> class Screen { ... };

const short shi = 40;
const short swi = 132;
```

- расширения типов:

```
Screen< shi, swi > bpObj2;  // расширения типа short до int
```

```
template <unsigned int size> Buf{ ... };
```

- преобразования целых типов:

```
Buf< 1024 > bpObj;  // преобразование из int в unsigned int
```

(Более подробно они описаны в разделе 9.3.)

```
extern void foo( char * );
extern void bar( void * );
typedef void (*PFV)( void * );
const unsigned int x = 1024;

template <class Type,
          unsigned int size,
          PFV handler> class Array { ... };

Array<int, 1024U, bar> a0;   // правильно: преобразование не нужно
Array<int, 1024U, foo> a1;   // ошибка: foo != PFV

Array<int, 1024, bar> a2;    // правильно: 1024 преобразуется в unsigned
          int
Array<int, 1024, bar> a3;    // ошибка: foo != PFV

Array<int, x, bar> a4;       // правильно: преобразование не нужно
```

Рассмотрим следующие объявления:

```
Array<int, x, foo> a5;       // ошибка: foo != PFV
```

Объекты `a0` и `a4` класса `Array` определены правильно, так как аргументы шаблона точно соответствуют типам параметров. Объект `a2` также определен правильно, потому что аргумент `1024` типа `int` приводится к типу `unsigned int` параметра-константы `size` с помощью преобразования целых типов. Объявления `a1`, `a3` и `a5` ошибочны, так как не существует преобразования между любыми двумя типами функций.

Приведение значения `0` целого типа к типу указателя недопустимо:

```
template <int *ptr>
class BufPtr { ... };

// ошибка: 0 имеет тип int
// неявное преобразование в нулевой указатель не применяется
BufPtr< 0 > nil;
```

Упражнение 16.3

Укажите, какие из данных конкретизированных шаблонов действительно приводят к конкретизации:

```
template < class Type >
   class Stack { };

void f1( Stack< char > );  // (a)

class Exercise {
   // ...
   Stack< double > &rsd;   // (b)
   Stack< int >    si;     // (c)
};

int main() {
   Stack< char > *sc;      // (d)
   f1( *sc );              // (e)

   int iObj = sizeof( Stack< string > );  // (f)
}
```

Упражнение 16.4

```
template < int *ptr > class Ptr ( ... };
template < class Type, int size > class Fixed_Array { ... };
```

Какие из следующих конкретизаций шаблонов корректны? Почему?

```
(a) const int size = 1024;
template < int hi, int wid > class Screen { ... };
```

```
(b) int arr[10];
    Ptr< arr > bp2;
    Ptr< &size > bp1;

 (c) Ptr < 0 > bp3;
```

```
(d) const int hi = 40;
    const int wi = 80;


(e) const int size_val = 1024;




(f) unsigned int fasize = 255;

    Screen< hi, wi+32 > sObj;

    Fixed_Array< string, size_val > fa1;


(g) const double db = 3.1415;

    Fixed_Array< int, fasize > fa2;

    Fixed_Array< double, db > fa3;
```

16.3. Функции-члены шаблонов классов

Как и для обычных классов, функция-член шаблона класса может быть определена либо внутри определения шаблона (и тогда называется встроенной), либо вне его. Мы уже встречались со встроенными функциями-членами при рассмотрении шаблона Queue. Например, конструктор Queue является встроенным, так как определен внутри

```
template <class Type>
class Queue {
   // ...
public:
   // встроенный конструктор
   Queue() : front( 0 ), back( 0 ) { }
   // ...
```

определения шаблона класса:

```
};
```

При определении функции-члена шаблона вне определения самого шаблона следует применять специальный синтаксис для обозначения того, членом какого именно шаблона является функция. Определению функции-члена должно предшествовать ключевое слово template, за которым следуют параметры шаблона. Так, конструктор Queue можно определить следующим образом:

```
template <class Type>
class Queue {
public:
   Queue();
private:
   // ...
};
template <class Type>
inline Queue<Type>::
   Queue( ) { front = back = 0; }
```

За первым вхождением Queue (перед оператором ::) следует список параметров, показывающий, какому шаблону принадлежит данная функция-член. Второе вхождение Queue в определение конструктора (после оператора ::) содержит имя функции-члена, за которым может следовать список параметров шаблона, хотя это и необязательно. После имени функции идет ее определение;. в нем могут быть ссылки на параметр шаблона Type всюду, где в определении обычной функции использовалось бы имя типа.

Функция-член шаблона класса сама является шаблоном. Стандарт C++ требует, чтобы она конкретизировалась только при вызове либо при взятии ее адреса. (Некоторые более старые компиляторы конкретизируют такие функции одновременно с конкретизацией самого шаблона класса.) При конкретизации функции-члена используется тип того объекта, для которого функция вызвана:

```
Queue<string> qs;
```

Объект qs имеет тип Queue<string>. При инициализации объекта этого класса вызывается конструктор Queue<string>. В данном случае аргументом, которым конкретизируется функция-член (конструктор), будет string.

Функция-член шаблона конкретизируется только при реальном использовании в программе (т.е. при вызове или взятии ее адреса). От того, в какой именно момент конкретизируется функция-член, зависит разрешение имен в ее определении (см. раздел 16.11) и объявление ее специализации (см. раздел 16.9).

16.3.1. Функции-члены шаблонов Queue и QueueItem

Чтобы понять, как определяются и используются функции-члены шаблонов классов, продолжим изучение шаблонов Queue и QueueItem:

```
template <class Type>
class Queue {
public:
   Queue() : front( 0 ), back ( 0 ) { }
   ~Queue();

   Type& remove();
   void add( const Type & );
   bool is_empty() const {
      return front == 0;
   }
private:
   QueueItem<Type> *front;
   QueueItem<Type> *back;
};
```

Деструктор, а также функции-члены `remove()` и `add()` определены не в теле шаблона, а

```
template <class Type>
Queue<Type>::~Queue()
{
   while (! is_empty() )
      remove();
}
```

вне его. Деструктор `Queue` опустошает очередь:

```
template <class Type>
void Queue<Type>::add( const Type &val )
{
   // создать новый объект QueueItem
   QueueItem<Type> *pt =
      new QueueItem<Type>( val );

   if ( is_empty() )
      front = back = pt;
   else
   {
      back->next = pt;
      back = pt;
   }
}
```

Функция-член `Queue<Type>::add()` помещает новый элемент в конец очереди:

Функция-член `Queue<Type>::remove()` возвращает значение элемента, находящегося в начале очереди, и удаляет сам элемент.

```
#include <iostream>
#include <cstdlib>

template <class Type>
Type Queue<Type>::remove()
{
   if ( is_empty() )
   {
      cerr << "remove() вызвана для пустой очереди\n";
      exit( -1 );
   }

   QueueItem<Type> *pt = front;
   front = front->next;

   Type retval = pt->item;
   delete pt;
   return retval;
}
```

Мы поместили определения функций-членов в заголовочный файл Queue.h, включив его в каждый файл, где возможны конкретизации функций. (Обоснование этого решения, а также рассмотрение более общих вопросов, касающихся модели компиляции шаблонов, мы отложим до раздела 16.8.)

В следующей программе иллюстрируется использование и конкретизация функции-члена

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

int main()
{
   // конкретизируется класс Queue<int>
   // оператор new требует, чтобы Queue<int> был определен
   Queue<int> *p_qi = new Queue<int>;

   int ival;
   for ( ival = 0; ival < 10; ++ival )
      // конкретизируется функция-член add()
      p_qi->add( ival );

   int err_cnt = 0;
   for ( ival = 0; ival < 10; ++ival ) {
      // конкретизируется функция-член remove()
      int qval = p_qi->remove();

      if ( ival != qval ) err_cnt++;
   }

   if ( !err_cnt )
      cout << "!! queue executed ok\n";
   else cerr << "?? queue errors: " << err_cnt << endl;
   return 0;
}
```

шаблона Queue:

После компиляции и запуска программа выводит следующую строку:

```
!! queue executed ok
```

Упражнение 16.5

Используя шаблон класса `Screen`, определенный в разделе 16.2, реализуйте функции-члены `Screen` (см. разделы 13.3, 13.4 и 13.6) в виде функций-членов шаблона.

16.4. Объявления друзей в шаблонах классов

- обычный (не шаблонный) дружественный класс или дружественная функция. В следующем примере функция `foo()`, функция-член `bar()` и класс `foobar`

```
class Foo {
   void bar();
};

template <class T>
class QueueItem {
   friend class foobar;
   friend void foo();
   friend void Foo::bar();
   // ...
```

объявлены друзьями всех конкретизаций шаблона `QueueItem`:

```
};
```

Ни класс `foobar`, ни функцию `foo()` не обязательно объявлять или определять в глобальной области видимости перед объявлением их друзьями шаблона `QueueItem`.

Однако перед тем как объявить другом какой-либо из членов класса `Foo`, необходимо определить его. Напомним, что член класса может быть введен в область видимости только через определение объемлющего класса. `QueueItem` не может ссылаться на `Foo::bar()`, пока не будет найдено определение `Foo`;

- *связанный* дружественный шаблон класса или функции. В следующем примере определено взаимно однозначное соответствие между классами, конкретизированными по шаблону `QueueItem`, и их друзьями – также конкретизациями шаблонов. Для каждого класса, конкретизированного по шаблону `QueueItem`, ассоциированные конкретизации `foobar`, `foo()` и

```
template <class Type>
   class foobar { ... };

template <class Type>
   void foo( QueueItem<Type> );

template <class Type>
class Queue {
      void bar();
      // ...
};

template <class Type>
class QueueItem {
   friend class foobar<Type>;
   friend void foo<Type>( QueueItem<Type> );
   friend void Queue<Type>::bar();

   // ...
```

`Queue::bar()` являются друзьями.

```
};
```

Прежде чем шаблон класса можно будет использовать в объявлениях друзей, он сам должен быть объявлен или определен. В нашем примере шаблоны классов `foobar` и `Queue`, а также шаблон функции `foo()` следует объявить до того, как они объявлены друзьями в `QueueItem`.

Синтаксис, использованный для объявления `foo()` другом, может показаться странным:

```
friend void foo<Type>( QueueItem<Type> );
```

За именем функции следует список явных аргументов шаблона: `foo<Type>`. Такой синтаксис показывает, что в качестве друга объявляется конкретизированный шаблон функции `foo()`. Если бы список явных аргументов был опущен:

```
friend void foo( QueueItem<Type> );
```

то компилятор интерпретировал бы объявление как относящееся к обычной функции (а не к шаблону), у которой тип параметра – это экземпляр шаблона `QueueItem`. Как отмечалось в разделе 10.6, шаблон функции и одноименная обычная функция могут сосуществовать, и присутствие объявления такого шаблона перед определением класса `QueueItem` не вынуждает компилятор соотнести объявление друга именно с ним. Для того, чтобы соотнесение было верным, в конкретизированном шаблоне функции необходимо указать список явных аргументов;

- *несвязанный* дружественный шаблон класса или функции. В следующем примере имеется отображение один-ко-многим между конкретизациями шаблона класса `QueueItem` и его друзьями. Для каждой конкретизации типа `QueueItem` все

```
template <class Type>
class QueueItem {
   template <class T>
      friend class foobar;

   template <class T>
      friend void foo( QueueItem<T> );

   template <class T>
      friend class Queue<T>::bar();

   // ...
```

конкретизации `foobar`, `foo()` и `Queue<T>::bar()` являются друзьями:

```
};
```

Следует отметить, что этот вид объявлений друзей в шаблоне класса не поддерживается компиляторами, написанными до принятия стандарта C++.

16.4.1. Объявления друзей в шаблонах Queue и QueueItem

Поскольку `QueueItem` не предназначен для непосредственного использования в вызывающей программе, то объявление конструктора этого класса помещено в закрытую

секцию шаблона. Теперь класс Queue необходимо объявить другом QueueItem, чтобы можно было создавать и манипулировать объектами последнего.

Существует два способа объявить шаблон класса другом. Первый заключается в том,

```
template <class Type>
class QueueItem {
   // любой экземпляр Queue является другом
   // любого экземпляра QueueItem
   template <class T> friend class Queue;
};
```

чтобы объявить любой экземпляр Queue другом любого экземпляра QueueItem:

Однако нет смысла объявлять, например, класс Queue, конкретизированный типом string, другом QueueItem, конкретизированного типом complex<double>. Queue<string> должен быть другом только для класса QueueItem<string>. Таким образом, нам нужно взаимно однозначное соответствие между экземплярами Queue и QueueItem, конкретизированными одинаковыми типами. Чтобы добиться этого,

```
template <class Type>
class QueueItem {
   // для любого экземпляра QueueItem другом является
   // только конкретизированный тем же типом экземпляр Queue
   friend class Queue<Type>;

   // ...
};
```

применим второй метод объявления друзей:

Данное объявление говорит о том, что для любой конкретизации QueueItem некоторым типом экземпляр Queue, конкретизированный тем же типом, является другом. Так, экземпляр Queue, конкретизированный типом int, будет другом экземпляра QueueItem, тоже конкретизированного типом int. Но для экземпляров QueueItem, конкретизированных типами complex<double> или string, этот экземпляр Queue другом не будет.

В любой точке программы у пользователю может понадобиться распечатать содержимое объекта Queue. Такая возможность предоставляется с помощью перегруженного оператора вывода. Этот оператор должен быть объявлен другом шаблона Queue, так как

```
// как задать аргумент типа Queue?
```

ему необходим доступ к закрытым членам класса. Какой же будет его сигнатура?

```
ostream& operator<<( ostream &, ??? );
```

Поскольку Queue – это шаблон класса, то в имени конкретизированного экземпляра должен быть задан полный список аргументов:

```
ostream& operator<<( ostream &, const Queue<int> & );
```

Так мы определили оператор вывода для класса, конкретизированного из шаблона Queue типом int. Но что, если Queue – это очередь элементов типа string?

```
ostream& operator<<( ostream &, const Queue<string> & );
```

Вместо того чтобы явно определять нужный оператор вывода по мере необходимости, желательно сразу определить общий оператор, который будет работать для любой конкретизации Queue. Например:

```
ostream& operator<<( ostream &, const Queue<Type> & );
```

```
template <class Type> ostream&
```

Однако из этого перегруженного оператора вывода придется сделать шаблон функции:

```
    operator<<( ostream &, const Queue<Type> & );
```

Теперь всякий раз, когда оператору ostream передается конкретизированный экземпляр Queue, конкретизируется и вызывается шаблон функции. Вот одна из возможных

```
template <class Type>
ostream& operator<<( ostream &os, const Queue<Type> &q )
{
   os << "< ";
   QueueItem<Type> *p;
   for ( p = q.front; p; p = p->next )
      os << *p << " ";
   os << " >";
   return os;
```

реализаций оператора вывода в виде такого шаблона:

```
}
```

Если очередь объектов типа int содержит значения 3, 5, 8, 13, то распечатка ее содержимого с помощью такого оператора дает

```
< 3 5 8 13 >
```

Обратите внимание, что оператор вывода обращается к закрытому члену front класса

```
template <class Type>
class Queue {
   friend ostream&
      operator<<( ostream &, const Queue<Type> & );
   // ...
```

Queue. Поэтому оператор необходимо объявить другом Queue:

```
};
```

Здесь, как и при объявлении друга в шаблоне класса Queue, создается взаимно однозначное соответствие между конкретизациями Queue и оператора operator<<().

Распечатка элементов Queue производится оператором вывода operator<<() класса QueueItem:

```
os << *p;
```

Этот оператор также должен быть реализован в виде шаблона функции; тогда можно

```
template <class Type>
ostream& operator<<( ostream &os, const QueueItem<Type> &qi )
{
   os << qi.item;
   return os;
}
```

быть уверенным, что в нужный момент будет конкретизирован подходящий экземпляр:

Поскольку здесь имеется обращение к закрытому члену item класса QueueItem, оператор

```
template <class Type>
class QueueItem {
   friend class Queue<Type>;
   friend ostream&
      operator<<( ostream &, const QueueItem<Type> & );
   // ...
};
```

следует объявить другом шаблона QueueItem. Это делается следующим образом:

Оператор вывода класса QueueItem полагается на то, что item умеет распечатывать себя:

```
os << qi.item;
```

Это порождает тонкую зависимость типов при конкретизации Queue. Любой определенный пользователем и связанный с Queue класс, содержимое которого нужно распечатывать, должен предоставлять оператор вывода. В языке нет механизма, с помощью которого можно было бы задать такую зависимость в определении самого шаблона Queue. Но если оператор вывода не определен для типа, с которым конкретизируется данный шаблон, и делается попытка вывести содержимое конкретизированного экземпляра, то в том месте, где используется отсутствующий оператор вывода, компилятор выдает сообщение об ошибке. Шаблон Queue можно конкретизировать типом, не имеющим оператора вывода, – при условии, что не будет попытки распечатать содержимое очереди.

Следующая программа демонстрирует конкретизацию и использование функций-друзей шаблонов классов Queue и QueueItem:

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

int main() {
   Queue<int> qi;
   // конкретизируются оба экземпляра
   //   ostream& operator<<(ostream &os, const Queue<int> &)
   //   ostream& operator<<(ostream &os, const QueueItem<int> &)
   cout << qi << endl;

   int ival;
   for ( ival = 0; ival < 10; ++ival )
      qi.add( ival );
   cout << qi << endl;

   int err_cnt = 0;
   for ( ival = 0; ival < 10; ++ival ) {
      int qval = qi.remove();
      if ( ival != qval ) err_cnt++;
   }

   cout << qi << endl;
   if ( !err_cnt )
      cout << "!! queue executed ok\n";
   else cout << "?? queue errors: " << err_cnt << endl;
   return 0;
}
```

После компиляции и запуска программа выдает результат:

```
< >
< 0 1 2 3 4 5 6 7 8 9 >
< >
!! queue executed ok
```

Упражнение 16.6

Пользуясь шаблоном класса Screen, определенным в упражнении 16.5, реализуйте операторы ввода и вывода (см. упражнение 15.6 из раздела 15.2) в виде шаблонов. Объясните, почему вы выбрали тот, а не иной способ объявления друзей класса Screen, добавленных в его шаблон.

16.5. Статические члены шаблонов класса

В шаблоне класса могут быть объявлены статические данные-члены. Каждый конкретизированный экземпляр имеет собственный набор таких членов. Рассмотрим операторы new() и delete() для шаблона QueueItem. В класс QueueItem нужно

```
static QueueItem<Type> *free_list;
```

добавить два статических члена:

```
static const unsigned QueueItem_chunk;
```

Модифицированное определение шаблона QueueItem выглядит так:

```
#include <cstddef>

template <class Type>
class QueueItem {
   // ...
private:
   void *operator new( size_t );
   void operator delete( void *, size_t );
   // ...
   static QueueItem *free_list;
   static const unsigned QueueItem_chunk;
   // ...
};
```

Операторы `new()` и `delete()` объявлены закрытыми, чтобы предотвратить создание объектов типа `QueueItem` вызывающей программой: это разрешается только членам и друзьям `QueueItem` (к примеру, шаблону `Queue`).

```
template <class Type> void*
   QueueItem<Type>::operator new( size_t size )
{
   QueueItem<Type> *p;
   if ( ! free_list )
   {
      size_t chunk = QueueItem_chunk * size;
      free_list = p =
      reinterpret_cast< QueueItem<Type>* >
                     ( new char[chunk] );

      for ( ; p != &free_list[ QueueItem_chunk - 1 ]; ++p )
         p->next = p + 1;
      p->next = 0;
   }

   p = free_list;
   free_list = free_list->next;
   return p;
}
```

Оператор `new()` можно реализовать таким образом:

```
template <class Type>
void QueueItem<Type>::
   operator delete( void *p, size_t )
{
   static_cast< QueueItem<Type>* >( p )->next = free_list;
   free_list = static_cast< QueueItem<Type>* > ( p );
}
```

А реализация оператора `delete()` выглядит так:

Теперь остается инициализировать статические члены `free_list` и `QueueItem_chunk`. Вот шаблон для определения статических данных-членов:

```
/* для каждой конкретизации QueueItem сгенерировать
 * соответствующий free_list и инициализировать его нулем
 */
template <class T>
   QueueItem<T> *QueueItem<T>::free_list = 0;

/* для каждой конкретизации QueueItem сгенерировать
 * соответствующий QueueItem_chunk и инициализировать его значением 24
 */
template <class T>
   const unsigned int

   QueueItem<T>::QueueItem_chunk = 24;
```

Определение шаблона статического члена должно быть вынесено за пределы определения самого шаблона класса, которое начинается с ключевого слово `template` с последующим списком параметров `<class T>`. Имени статического члена предшествует префикс `QueueItem<T>::`, показывающий, что этот член принадлежит именно шаблону `QueueItem`. Определения таких членов помещаются в заголовочный файл `Queue.h` и должны включаться во все файлы, где производится их конкретизация. (В разделе 16.8 мы объясним, почему решили делать именно так, и затронем другие вопросы, касающиеся модели компиляции шаблонов.)

Статический член конкретизируется по шаблону только в том случае, когда реально используется в программе. Сам такой член тоже является шаблоном. Определение шаблона для него не приводит к выделению памяти: она выделяется только для конкретизированного экземпляра статического члена. Каждая подобная конкретизация соответствует конкретизации шаблона класса. Таким образом, обращение к экземпляру статического члена всегда производится через некоторый конкретизированный экземпляр

```
// ошибка: QueueItem - это не реальный конкретизированный экземпляр
int ival0 = QueueItem::QueueItem_chunk;

int ival1 = QueueItem<string>::QueueItem_chunk;  // правильно
```

класса:

```
int ival2 = QueueItem<int>::QueueItem_chunk;     // правильно
```

Упражнение 16.7

Реализуйте определенные в разделе 15.8 операторы `new()` и `delete()` и относящиеся к ним статические члены `screenChunk` и `freeStore` для шаблона класса `Screen`, построенного в упражнении 16.6.

16.6. Вложенные типы шаблонов классов

Шаблон класса `QueueItem` применяется только как вспомогательное средство для реализации `Queue`. Чтобы запретить любое другое использование, в шаблоне `QueueItem` имеется закрытый конструктор, позволяющий создавать объекты этого класса исключительно функциям-членам класса `Queue`, объявленным друзьями `QueueItem`. Хотя шаблон `QueueItem` виден во всей программе, создать объекты этого класса или обратиться к его членам можно только при посредстве функций-членов `Queue`.

Альтернативный подход к реализации состоит в том, чтобы вложить определение шаблона класса QueueItem в закрытую секцию шаблона Queue. Поскольку QueueItem является вложенным закрытым типом, он становится недоступным вызывающей программе, и обратиться к нему можно лишь из шаблона класса Queue и его друзей (например, оператора вывода). Если же сделать члены QueueItem открытыми, то объявлять Queue другом QueueItem не понадобится.

Семантика исходной реализации при этом сохраняется, но отношение между шаблонами QueueItem и Queue моделируется более элегантно.

Поскольку при любой конкретизации шаблона Queue требуется конкретизировать тем же типом и QueueItem, то вложенный класс должен быть шаблоном. Вложенные классы шаблонов сами являются шаблонами классов, а параметры объемлющего шаблона можно

```
template <class Type>
class Queue:
   // ...
private:
   class QueueItem {
   public:
      QueueItem( Type val )
              : item( val ), next( 0 ) { ... }

      Type item;
      QueueItem *next;
   };
   // поскольку QueueItem - вложенный тип,
   // а не шаблон, определенный вне Queue,
   // то аргумент шаблона <Type> после QueueItem можно опустить
   QueueItem *front, *back;
   // ...
```

использовать во вложенном:

```
};
```

При каждой конкретизации Queue создается также класс QueueItem с подходящим аргументом для Type. Между конкретизациями шаблонов QueueItem и Queue имеется взаимно однозначное соответствие.

Вложенный в шаблон класс конкретизируется только в том случае, если он используется в контексте, где требуется полный тип класса. В разделе 16.2 мы упоминали, что конкретизация шаблона класса Queue типом int не означает автоматической конкретизации и класса QueueItem<int>. Члены front и back – это указатели на QueueItem<int>, а если объявлены только указатели на некоторый тип, то конкретизировать соответствующий класс не обязательно, хотя QueueItem вложен в шаблон класса Queue. QueueItem<int> конкретизируется только тогда, когда указатели front или back разыменовываются в функциях-членах класса Queue<int>.

Внутри шаблона класса можно также объявлять перечисления и определять типы (с помощью typedef):

```
template <class Type, int size>
class Buffer:
public:
   enum Buf_vals { last = size-1, Buf_size };
   typedef Type BufType;
   BufType array[ size ];
   // ...
}
```

Вместо того чтобы явно включать член Buf_size, в шаблоне класса Buffer объявляется перечисление с двумя элементами, которые инициализируются значением параметра шаблона. Например, объявление

```
Buffer<int, 512> small_buf;
```

устанавливает Buf_size в 512, а last – в 511. Аналогично

```
Buffer<int, 1024> medium_buf;
```

устанавливает Buf_size в 1024, а last – в 1023.

Открытый вложенный тип разрешается использовать и вне определения объемлющего класса. Однако вызывающая программа может ссылаться лишь на конкретизированные экземпляры подобного типа (или элементов вложенного перечисления). В таком случае имени вложенного типа должно предшествовать имя конкретизированного шаблона

```
// ошибка: какая конкретизация Buffer?
Buffer::Buf_vals bfv0;
```

класса:

```
Buffer<int,512>::Buf_vals bfv1;  // правильно
```

Это правило применимо и тогда, когда во вложенном типе не используются параметры включающего шаблона:

```
template <class T> class Q {
public:
   enum QA { empty, full };   // не зависит от параметров
   QA status;
   // ...
};

#include <iostream>

int main() {
   Q<double> qd;
   Q<int> qi;

   qd.status = Q::empty;  // ошибка: какая конкретизация Q?
   qd.status = Q<double>::empty;  // правильно

   int val1 = Q<double>::empty;
   int val2 = Q<int>::empty;
   if ( val1 != val2 )
      cerr << "ошибка реализации!" << endl;
   return 0;
}
```

Во всех конкретизациях `Q` значения `empty` одинаковы, но при ссылке на `empty` необходимо указывать, какому именно экземпляру `Q` принадлежит перечисление.

Упражнение 16.8

Определите класс `List` и вложенный в него `ListItem` из раздела 13.10 как шаблоны. Реализуйте аналогичные определения для ассоциированных членов класса.

16.7. Шаблоны-члены

Шаблон функции или класса может быть членом обычного класса или шаблона класса. Определение шаблона-члена похоже на определение шаблона: ему предшествует ключевое слово `template`, за которым идет список параметров:

```
template <class T>
class Queue {
private:
   // шаблон класса-члена
   template <class Type>
      class CL
   {
      Type member;
      T mem;
   };
   // ...
public:
   // шаблон функции-члена
   template <class Iter>
      void assign( Iter first, Iter last )
   {
      while ( ! is_empty() )
         remove();     // вызывается Queue<T>::remove()

      for ( ; first != last; ++first )
          add( *first );  // вызывается Queue<T>::add( const T & )
   }

}
```

(Отметим, что шаблоны-члены не поддерживаются компиляторами, написанными до принятия стандарта C++. Эта возможность была добавлена в язык для поддержки реализации абстрактных контейнерных типов, представленных в главе 6.)

Объявление шаблона-члена имеет собственные параметры. Например, у шаблона класса `CL` есть параметр `Type`, а у шаблона функции `assign()` – параметр `Iter`. Помимо этого, в определении шаблона-члена могут использоваться параметры объемлющего шаблона класса. Например, у шаблона `CL` есть член типа `T`, представляющего параметр включающего шаблона `Queue`.

Объявление шаблона-члена в шаблоне класса `Queue` означает, что конкретизация `Queue` потенциально может содержать бесконечное число различных вложенных классов `CL` функций-членов `assign()`. Так, конкретизированный экземпляр `Queue<int>` включает

```
Queue<int>::CL<char>
```

вложенные типы:

```
Queue<int>::CL<string>
```

```
void Queue<int>::assign( int *, int * )
void Queue<int>::assign( vector<int>::iterator,
```

и вложенные функции:

```
                         vector<int>::iterator )
```

Для шаблона-члена действуют те же правила доступа, что и для других членов класса. Так как шаблон `CL` является закрытым членом шаблона `Queue`, то лишь функции-члены и друзья `Queue` могут ссылаться на его конкретизации. С другой стороны, шаблон функции `assign()` объявлен открытым членом и, значит, доступен во всей программе.

Шаблон-член конкретизируется при его использовании в программе. Например,

```
int main()
{
   // конкретизация Queue<int>
   Queue<int> qi;

   // конкретизация Queue<int>::assign( int *, int * )
   int ai[4] = { 0, 3, 6, 9 };
   qi.assign( ai, ai + 4 );

   // конкретизация Queue<int>::assign( vector<int>::iterator,
   //                                   vector<int>::iterator )
   vector<int> vi( ai, ai + 4 );
   qi.assign( vi.begin(), vi.end() );
```

assign() конкретизируется в момент обращения к ней из main():

```
}
```

Шаблон функции assign(), являющийся членом шаблона класса Queue, иллюстрирует необходимость применения шаблонов-членов для поддержки контейнерных типов. Предположим, имеется очередь типа Queue<int>, в которую нужно поместить содержимое любого другого контейнера (списка, вектора или обычного массива), причем его элементы имеют либо тип int (т.е. тот же, что у элементов очереди), либо приводимый к типу int. Шаблон-член assign()позволяет это сделать. Поскольку может быть использован любой контейнерный тип, то интерфейс assign() программируется в расчете на употребление итераторов; в результате реализация оказывается не зависящей от фактического типа, на который итераторы указывают.

В функции main() шаблон-член assign() сначала конкретизируется типом int*, что позволяет поместить в qi содержимое массива элементов типа int. Затем шаблон-член конкретизируется типом vector<int>::iterator – это дает возможность поместить в очередь qi содержимое вектора элементов типа int. Контейнер, содержимое которого помещается в очередь, не обязательно должен состоять из элементов типа int. Разрешен любой тип, который приводится к int. Чтобы понять, почему это так, еще раз посмотрим

```
template <class Iter>
      void assign( Iter first, Iter last )
{
   // удалить все элементы из очереди

   for ( ; first != last; ++first )
      add( *first );
```

на определение assign():

```
}
```

Вызываемая из assign() функция add() – это функция-член Queue<Type>::add(). Если Queue конкретизируется типом int, то у add() будет следующий прототип:

```
void Queue<int>::add( const int &val );
```

Аргумент *first должен иметь тип int либо тип, которым можно инициализировать параметр-ссылку на const int. Преобразования типов допустимы. Например, если

воспользоваться классом SmallInt из раздела 15.9, то содержимое контейнера, в котором хранятся элементы типа SmallInt, с помощью шаблона-члена assign() помещается в очередь типа Queue<int>. Это возможно потому, что в классе SmallInt имеется

```
class SmallInt {
public:
   SmallInt( int ival = 0 ) : value( ival ) { }

   // конвертер: SmallInt ==> int
   operator int() { return value; }

   // ...
private:
   int value;
};

int main()
{
   // конкретизация Queue<int>
   Queue<int> qi;

   vector<SmallInt> vsi;
   // заполнить вектор
   // конкретизация
   // Queue<int>::assign( vector<SmallInt>::iterator,
   //                     vector<SmallInt>::iterator )
   qi.assign( vsi.begin(), vsi.end() );

   list<int*> lpi;
   // заполнить список

   // ошибка при конкретизации шаблона-члена assign():
   // нет преобразования из int* в int
   qi.assign( lpi.begin(), lpi.end() );
```

конвертер для приведения SmallInt к int:

```
}
```

Первая конкретизация assign() правильна, так как существует неявное преобразование из типа SmallInt в тип int и, следовательно, обращение к add() корректно. Вторая же конкретизация ошибочна: объект типа int* не может инициализировать ссылку на тип const int, поэтому вызвать функцию add() невозможно.

Для контейнерных типов из стандартной библиотеки C++ имеется функция assign(), которая ведет себя так же, как функция-шаблон assign() для нашего класса Queue.

Любую функцию-член можно задать в виде шаблона. Это относится, в частности, к конструктору. Например, для шаблона класса Queue его можно определить следующим образом:

```
template <class T>
class Queue {
   // ...
public:
   // шаблон-член конструктора
   template <class Iter>
   Queue( Iter first, Iter last )
        : front( 0 ), back( 0 )
   {
      for ( ; first != last; ++first )
          add( * first );
   }
};
```

Такой конструктор позволяет инициализировать очередь содержимым другого контейнера. У контейнерных типов из стандартной библиотеки C++ также есть предназначенные для этой цели конструкторы в виде шаблонов-членов. Кстати, в первом (в данном разделе) определении функции `main()` использовался конструктор-шаблон для вектора:

```
vector<int> vi( ai, ai + 4 );
```

Это определение конкретизирует шаблон конструктора для контейнера `vector<int>` типом `int*`, что позволяет инициализировать вектор содержимым массива элементов типа `int`.

Шаблон-член, как и обычные члены, может быть определен вне определения объемлющего класса или шаблона класса. Так, являющиеся членами шаблон класса `CL` или шаблон функции `assign()` могут быть следующим образом определены вне шаблона `Queue`:

```
template <class T>
class Queue {
private:
   template <class Type> class CL;
   // ...
public:
   template <class Iter>
      void assign( Iter first, Iter last );
   // ...
};

template <class T> template <class Type>
   class Queue<T>::CL<Type>
{
   Type member;
   T mem;
};

template <class T> template <class Iter>
   void Queue<T>::assign( Iter first, Iter last )
{
   while ( ! is_empty() )
      remove();

   for ( ; first != last; ++first )
      add( *first );

}
```

Определению шаблона-члена, которое находится вне определения объемлющего шаблона класса, предшествует список параметров объемлющего шаблона класса, а за ним должен следовать собственный такой список. Вот почему определение шаблона функции `assign()` (члена шаблона класса `Queue`) начинается с

```
template <class T> template <class Iter>
```

Первый список параметров шаблона `template <class T>` относится к шаблону класса `Queue`. Второй – к самому шаблону-члену `assign()`. Имена параметров не обязаны совпадать с теми, которые указаны внутри определения объемлющего шаблона класса. Приведенная инструкция по-прежнему определяет шаблон-член `assign()`:

```
void Queue<TT>::assign( IterType first, IterType last )
```

```
template <class TT> template <class IterType>
{ ... }
```

16.8. Шаблоны классов и модель компиляции A

Определение шаблона класса – это лишь предписание для построения бесконечного множества типов классов. Сам по себе шаблон не определяет никакого класса. Например, когда компилятор видит:

```
template <class Type>
   class Queue { ... };
```

он только сохраняет внутреннее представление Queue. Позже, когда встречается реальное

```
int main() {
   Queue<int> *p_qi = new Queue<int>;
}
```

использование класса, конкретизированного по шаблону, скажем:

компилятор конкретизирует тип класса Queue<int>, применяя сохраненное внутреннее представление определения шаблона Queue.

Шаблон конкретизируется только тогда, когда он употребляется в контексте, требующем полного определения класса. (Этот вопрос подробно обсуждался в разделе 16.2.) В примере выше класс Queue<int> конкретизируется, потому что компилятор должен знать размер типа Queue<int>, чтобы выделить нужный объем памяти для объекта, созданного оператором new.

Компилятор может конкретизировать шаблон только тогда, когда он видел не только его объявление, но и фактическое определение, которое должно предшествовать тому месту

```
// объявление шаблона класса
template <class Type>
   class Queue;

Queue<int>* global_pi = 0;  // правильно: определение класса не нужно

int main() {
   // ошибка: необходима конкретизация
   //         определение шаблона класса должно быть видимо
   Queue<int> *p_qi = new Queue<int>;
}
```

программы, где этот шаблон используется:

Шаблон класса можно конкретизировать одним и тем же типом в нескольких файлах. Как и в случае с типами классов, когда определение класса должно присутствовать в каждом файле, где используются его члены, компилятор конкретизирует шаблон некоторым типом во всех файлах, в которых данный экземпляр употребляется в контексте, требующем полного определения класса. Чтобы определение шаблона было доступно везде, где может понадобиться конкретизация, его следует поместить в заголовочный файл.

Функции-члены и статические данные-члены шаблонов классов, а также вложенные в них типы ведут себя почти так же, как сами шаблоны. Определения членов шаблона используются для порождения экземпляров членов в конкретизированном шаблоне. Если компилятор видит:

```
template <class Type>
void Queue<Type>::add( const Type &val )

   { ... }
```

он сохраняет внутреннее представление `Queue<Type>::add()`. Позже, когда в программе встречается фактическое употребление этой функции-члена, допустим через объект типа `Queue<int>`, компилятор конкретизирует `Queue<int>::add(const int &)`, пользуясь

```
#include "Queue.h"

int main() {
   // конкретизация Queue<int>
   Queue<int> *p_qi = new Queue<int>;
   int ival;
   // ...
   // конкретизация Queue<int>::add( const int & )
   p_qi->add( ival );
   // ...
```

таким представлением:

```
}
```

Конкретизация шаблона класса некоторым типом не приводит к автоматической конкретизации всех его членов тем же типом. Член конкретизируется только при использовании в таком контексте, где необходимо его определение (т.е. вложенный тип употреблен так, что требуется его полное определение; вызвана функция-член или взят ее адрес; имеется обращение к значению статического члена).

Конкретизация функций-членов и статических членов шаблонов класса поднимает те же вопросы, которые мы уже обсуждали для шаблонов функций в разделе 10.5. Чтобы компилятор мог конкретизировать функцию-член или статический член шаблона класса, должно ли определение члена быть видимым в момент конкретизации? Например, должно ли определение функции-члена `add()` появиться до ее конкретизации типом `int` в `main()`? Следует ли помещать определения функций-членов и статических членов шаблонов класса в заголовочные файлы (как мы поступаем с определениями встроенных функций), которые включаются всюду, где применяются их конкретизированные экземпляры? Или конкретизации определения шаблона достаточно для того, чтобы этими членами можно было пользоваться, так что определения членов можно оставлять в файлах с исходными текстами (где обычно располагаются определения невстроенных функций-членов и статических членов)?

Для ответа на эти вопросы нам придется вспомнить *модель компиляции шаблонов* в C++, где формулируются требования к организации программы, в которой определяются и употребляются шаблоны. Обе модели (с включением и с разделением), описанные в разделе 10.5, в полной мере применимы и к определениям функций-членов и статических данных-членов шаблонов классов. В оставшейся части этого раздела описываются обе модели и объясняется их использование с определениями членов.

16.8.1. Модель компиляции с включением

В этой модели мы включаем определения функций-членов и статических членов шаблонов классов в каждый файл, где они конкретизируются. Для встроенных функций-членов, определенных в теле шаблона, это происходит автоматически. В противном

случае такое определение следует поместить в один заголовочный файл с определением шаблона класса. Именно этой моделью мы и пользуемся в настоящей книге. Например, определения шаблонов `Queue` и `QueueItem`, как и их функций-членов и статических членов, находятся в заголовочном файле `Queue.h`.

Подобное размещение не лишено недостатков: определения функций-членов могут быть довольно большими и содержать детали реализации, которые неинтересны пользователям или должны быть скрыты от них. Кроме того, многократная компиляция одного определения шаблона при обработке разных файлов увеличивает общее время компиляции программы. Описанная модель (если она доступна) позволяет отделить интерфейс шаблона от реализации (т.е. от определений функций-членов и статических данных-членов).

16.8.2. Модель компиляции с разделением

В этой модели определение шаблона класса и определения встроенных функций-членов помещаются в заголовочный файл, а определения невстроенных функций-членов и статических данных-членов – в файл с исходным текстом программы. Иными словами, определения шаблона класса и его членов организованы так же, как определения

```
// ---- Queue.h ----
// объявляет Queue как экспортируемый шаблон класса
export template <class Type>
class Queue {
   // ...
public:
   Type& remove();
   void add( const Type & );
   // ...
```

обычных классов (не шаблонов) и их членов:

```
// ---- Queue.C ----
// экспортированное определение шаблона класса Queue
// находится в Queue.h
#include "Queue.h"

template <class Type>
   void Queue<Type>::add( const Type &val ) { ... }

template <class Type>

};

   Type& Queue<Type>::remove() { ... }
```

Программа, в которой используется конкретизированная функция-член, должна перед конкретизацией включить заголовочный файл:

```
// ---- User.C ----
#include "Queue.h"

int main() {
   // конкретизация Queue<int>
   Queue<int> *p_qi = new Queue<int>;
   int ival;
   // ...
   // правильно: конкретизация Queue<int>::add( const int & )
   p_qi->add( ival );
   // ...
}
```

Хотя определение шаблона для функции-члена `add()` не видно в файле `User.C`, конкретизированный экземпляр `Queue<int>::add(const int &)` вызывать оттуда можно. Но для этого шаблон класса необходимо объявить *экспортируемым*.

Если он экспортируется, то для использования конкретизированных функций-членов или статических данных-членов необходимо знать лишь определение самого шаблона. Определения членов могут отсутствовать в тех файлах, где они конкретизируются.

Чтобы объявить шаблон класса экспортируемым, перед словом `template` в его

```
export template <class Type>
```

определении или объявлении нужно поставить ключевое слово `export`:

```
class Queue { ... };
```

В нашем примере слово `export` применено к шаблону класса `Queue` в файле `Queue.h`; этот файл включен в файл `Queue.C`, содержащий определения функций-членов `add()` и `remove()`, которые автоматически становятся экспортируемыми и не должны присутствовать в других файлах перед конкретизацией.

Отметим, что, хотя шаблон класса объявлен экспортируемым, его собственное определение должно присутствовать в файле `User.C`. Конкретизация `Queue<int>::add()` в `User.C` вводит определение класса, в котором объявлены функции-члены `Queue<int>::add()` и `Queue<int>::remove()`. Эти объявления обязаны предшествовать вызову указанных функций. Таким образом, слово `export` влияет лишь на обработку функций-членов и статических данных-членов.

Экспортируемыми можно объявлять также отдельные члены шаблона. В этом случае ключевое слово `export` указывается не перед шаблоном класса, а только перед экспортируемыми членами. Например, если автор шаблона класса `Queue` хочет экспортировать лишь функцию-член `Queue<Type>::add()` (т.е. изъять из заголовочного файла `Queue.h` только ее определение), то слово `export` можно указать именно в определении функции-члена `add()`:

```
// ---- Queue.h ----
template <class Type>
class Queue {
   // ...
public:
   Type& remove();
   void add( const Type & );
   // ...
};

// необходимо, так как remove() не экспортируется
template <class Type>
```

```
// ---- Queue.C ----
#include "Queue.h"

// экспортируется только функция-член add()
export template <class Type>

   Type& Queue<Type>::remove() { ... }

   void Queue<Type>::add( const Type &val ) { ... }
```

Обратите внимание, что определение шаблона для функции-члена `remove()` перенесено в заголовочный файл `Queue.h`. Это необходимо, поскольку `remove()` более не находится в экспортируемом шаблоне и, следовательно, ее определение должно быть видно во всех файлах, где вызываются конкретизированные экземпляры.

Определение функции-члена или статического члена шаблона объявляется экспортируемым только один раз во всей программе. Поскольку компилятор обрабатывает файлы последовательно, он обычно не в состоянии определить, что эти члены объявлены экспортируемыми в нескольких исходных файлах. В таком случае результаты могут быть следующими:

- при редактировании связей возникает ошибка, показывающая, что один и тот же член шаблона класса определен несколько раз;
- компилятор неоднократно конкретизирует некоторый член одним и тем же множеством аргументов шаблона, что приводит к ошибке повторного определения во время связывания программы;
- компилятор конкретизирует член с помощью одного из экспортированных определений шаблона, игнорируя все остальные.

Следовательно, нельзя утверждать, что при наличии в программе нескольких определений экспортированного члена шаблона обязательно будет сгенерирована ошибка. Создавая программу, надо быть внимательным и следить за тем, чтобы определения членов находились только в одном исходном файле.

Модель с разделением позволяет отделить интерфейс шаблона класса от его реализации и организовать программу так, что эти интерфейсы помещаются в заголовочные файлы, а реализации – в файлы с исходным текстом. Однако не все компиляторы поддерживают данную модель, а те, которые поддерживают, не всегда делают это правильно: для этого требуется более изощренная среда программирования, которая доступна не во всех реализациях C++.

В нашей книге используется только модель с включением, так как примеры работы с шаблонами небольшие и хотелось, чтобы они компилировались максимально большим числом компиляторов.

16.8.3. Явные объявления конкретизации

При использовании модели с включением определение члена шаблона класса помещается в каждый исходный файл, где может употребляться конкретизированный экземпляр. Точно неизвестно, где и когда компилятор конкретизирует такое определение, и некоторые компиляторы (особенно более старые) конкретизируют определение члена данным множеством аргументов шаблона неоднократно. Для использования в программе (на этапе сборки или на одной из предшествующих ей стадий) выбирается один из полученных экземпляров, а остальные игнорируются.

Результат работы программы не зависит от того, сколько раз конкретизировался шаблон: в конечном итоге употребляется лишь один экземпляр. Однако, если приложение состоит из большого числа файлов и некоторый шаблон конкретизируется в каждом из них, то время компиляции заметно возрастает.

Подобные проблемы, характерные для старых компиляторов, затрудняли использование шаблонов. Чтобы помочь программисту управлять моментом, когда конкретизация происходит, в стандарте C++ введено понятие *явного объявления конкретизации*, где за ключевым словом `template` идет слово `class` и имя конкретизируемого шаблона класса.

В следующем примере явно объявляется конкретизация шаблона `Queue<int>`, в котором

```
#include "Queue.h"

// явное объявление конкретизации
```

запрашивается конкретизация аргументом `int` шаблона класса `Queue`:

```
template class Queue<int>;
```

Если шаблон класса конкретизируется явно, то явно конкретизируются и все его члены, причем тем же типом аргумента. Следовательно, в файле, где встречается явное объявление, должно присутствовать не только определение шаблона, но и определения

```
template <class Type>
   class Queue;

// ошибка: шаблон Queue и его члены не определены
```

всех его членов. В противном случае выдается сообщение об ошибке:

```
template class Queue<int>;
```

Если в некотором исходном файле встречается явное объявление конкретизации, то что произойдет в других файлах, где используется такой же конкретизированный шаблон? Как сказать компилятору, что явное объявление имеется в другом файле и что при употреблении шаблона класса или его членов в этом файле конкретизировать ничего не надо?

Здесь, как и при использовании шаблонов функций (см. раздел 10.5.3), необходимо применить опцию компилятора, подавляющую неявные конкретизации. Эта опция

вынуждает компилятор предполагать, что все конкретизации шаблонов будут объявляться явно.

Упражнение 16.9

Куда бы вы поместили определения функций-членов и статических данных-членов своих шаблонов классов, если имеющийся у вас компилятор поддерживает модель компиляции с разделением? Объясните почему.

Упражнение 16.10

Имеется шаблон класса Screen, разработанный в упражнениях из предыдущих разделов (в том числе функции-члены, определенные в упражнении 16.5 из раздела 16.3, и статические члены, определенные в упражнении 16.7 из раздела 16.5). Организуйте программу так, чтобы воспользоваться преимуществами модели компиляции с разделением.

16.9. Специализации шаблонов классов A

Прежде чем приступать к рассмотрению специализаций шаблонов классов и причин, по которым в них может возникнуть надобность, добавим в шаблон Queue функции-члены min() и max(). Они будут обходить все элементы очереди и искать среди них соответственно минимальное и максимальное значения (правильнее, конечно, использовать для этой цели обобщенные алгоритмы min() и max(), представленные в главе 12, но мы определим эти функции как члены шаблона Queue, чтобы познакомиться

```
template <class Type>
class Queue {
   // ...
public:
   Type min();
   Type max();
   // ...
};

// найти минимальное значение в очереди Queue
template <class Type>
   Type Queue<Type>::min()
{
   assert( ! is_empty() );
   Type min_val = front->item;
   for ( QueueItem *pq = front->next; pq != 0; pq = pq->next )
      if ( pq->item < min_val )
         min_val = pq->item;
   return min_val;
}
// найти максимальное значение в очереди Queue
template <class Type>
   Type Queue<Type>::max()
{
   assert( ! is_empty() );
   Type max_val = front->item;
   for ( QueueItem *pq = front->next; pq != 0; pq = pq->next )
      if ( pq->item > max_val )
         max_val = pq->item;
   return max_val;
```

со специализациями.)

```
}
```

Следующая инструкция в функции-члене min() сравнивает два элемента очереди Queue:

```
pq->item < min_val
```

Здесь неявно присутствует требование к типам, которыми может конкретизироваться шаблон класса Queue: такой тип должен либо иметь возможность пользоваться предопределенным оператором “меньше” для встроенных типов, либо быть классом, в котором определен оператор operator<(). Если же этого оператора нет, то попытка применить min() к очереди приведет к ошибке компиляции в том месте, где вызывается несуществующий оператор сравнения. (Аналогичная проблема существует и в max(), только касается оператора operator>()).

```
class LongSouble {
public:
   LongDouble( double dbval ) : value( dval ) { }
   bool compareLess( const LongDouble & );
private:
   double value;
```

Предположим, что шаблон класса Queue нужно конкретизировать таким типом:

```
};
```

Но в этом классе нет оператора operator<(), позволяющего сравнивать два значения типа LongDouble, поэтому использовать для очереди типа Queue<LongDouble> функции-члены min() и max() нельзя. Одним из решений этой проблемы может стать определение глобальных operator<() и operator>(), в которых для сравнения значений типа Queue<LongDouble> используется функция-член compareLess. Эти глобальные операторы вызывались бы из min() и max() автоматически при сравнении объектов из очереди.

Однако мы рассмотрим другое решение, связанное со специализацией шаблонов класса: вместо общих определений функций-членов min() и max() при конкретизации шаблона Queue типом LongDouble мы определим специальные экземпляры Queue<LongDouble>::min() и Queue<LongDouble>::max(), основанные на функции-члене compareLess() класса LongDouble.

Это можно сделать, если воспользоваться *явным определением специализации*, где после ключевого слова template идет пара угловых скобок <>, а за ней – определение специализации члена класса. В приведенном примере для функций-членов min() и max() класса Queue<LongDouble>, конкретизированного из шаблона, определены явные специализации:

```
// определения явных специализаций
template<> LongDouble Queue<LongDouble>::min()
{
   assert( ! is_empty() );
   LongDouble min_val = front->item;
   for ( QueueItem *pq = front->next; pq != 0; pq = pq->next )
      if ( pq->item.compareLess( min_val ) )
         min_val = pq->item;
   return min_val;
}

template<> LongDouble Queue<LongDouble>::max()
{
   assert( ! is_empty() );
   LongDouble max_val = front->item;
   for ( QueueItem *pq = front->next; pq != 0; pq = pq->next )
      if ( max_val.compareLess( pq->item ) )
         max_val = pq->item;
   return max_val;
}
```

Хотя тип класса `Queue<LongDouble>` конкретизируется по шаблону, в каждом объекте этого типа используются специализированные функции-члены `min()` и `max()` – не те, что конкретизируются по обобщенным определениям этих функций в шаблоне класса `Queue`.

Поскольку определения явных специализаций `min()` и `max()` – это определения невстроенных функций, помещать их в заголовочный файл нельзя: они обязаны находится в файле с текстом программы. Однако явную специализацию функции можно

```
// объявления явных специализаций функций-членов
template <> LongDouble Queue<LongDouble>::min();
```

объявить, не определяя. Например:

```
template <> LongDouble Queue<LongDouble>::max();
```

Поместив эти объявления в заголовочный файл, а соответствующие определения – в исходный, мы можем организовать код так же, как и для определений функций-членов обычного класса.

Иногда определение всего шаблона оказывается непригодным для конкретизации некоторым типом. В таком случае программист может специализировать шаблон класса целиком. Напишем полное определение класса `Queue<LongDouble>`:

```
// QueueLD.h: определяет специализацию класса Queue<LongDouble>
#include "Queue.h"

template<> Queue<LongDouble> {
   Queue<LongDouble>();
   ~Queue<LongDouble>();

   LongDouble& remove();
   void add( const LongDouble & );
   bool is_empty() const;
   LongDouble min();
   LongDouble max();
private:
   // Некоторая реализация
};
```

Явную специализацию шаблона класса можно определять только после того, как общий шаблон уже был объявлен (хотя и не обязательно определен). Иными словами, должно быть известно, что специализируемое имя обозначает шаблон класса. Если в приведенном примере не включить заголовочный файл `Queue.h` перед определением явной специализации шаблона, компилятор выдаст сообщение об ошибке, указывая, что `Queue` – это не имя шаблона.

Если мы определяем специализацию всего шаблона класса, то должны определить также все без исключения функции-члены и статические данные-члены. Определения членов из общего шаблона никогда не используются для создания определений членов явной специализации: множества членов этих шаблонов могут различаться. Чтобы предоставить определение явной специализации для типа класса `Queue<LongDouble>`, придется определить не только функции-члены `min()` и `max()`, но и все остальные.

Если класс специализируется целиком, лексемы `template<>` помещаются только перед

```
#include "QueueLD.h"

// определяет функцию-член min()
// из специализированного шаблона класса
```

определением явной специализации всего шаблона:

```
LongDouble Queue<LongDouble>::min() { }
```

Класс не может в одних файлах конкретизироваться из общего определения шаблона, а в других – из специализированного, если задано одно и то же множество аргументов. Например, специализацию шаблона `QueueItem<LongDouble>` необходимо объявлять в

```
// ---- File1.C ----
#include "Queue.h"

void ReadIn( Queue<LongDouble> *pq ) {
   // использование pq->add()
   // приводит к конкретизации QueueItem<LongDouble>
}
```

каждом файле, где она используется:

```
// ---- File2.C ----
#include "QueueLD.h"

void ReadIn( Queue<LongDouble> * );

int main() {
   // используется определение специализации для Queue<LongDouble>
   Queue<LongDouble> *qld = new Queue<LongDouble>;

   ReadIn( qld );
   // ...
}
```

Эта программа некорректна, хотя большинство компиляторов ошибку не обнаружат: заголовочный файл QueueLD.h следует включать во все файлы, где используется Queue<LongDouble>, причем до первого использования.

16.10. Частичные специализации шаблонов классов A

Если у шаблона класса есть несколько параметров, то можно специализировать его только для одного или нескольких аргументов, оставляя другие неспециализированными. Иными словами, допустимо написать шаблон, соответствующий общему во всем, кроме тех параметров, вместо которых подставлены фактические типы или значения. Такой механизм носит название *частичной специализации* шаблона класса. Она может понадобиться при определении реализации, более подходящей для конкретного набора аргументов.

Рассмотрим шаблон класса Screen, введенный в разделе 16.2. Частичная специализации

```
template <int hi, int wid>
class Screen {
   // ...
};

// частичная специализация шаблона класса Screen
template <int hi>
class Screen<hi, 80> {
public:
   Screen();
   // ...
private:
   string            _screen;
   string::size_type _cursor;
   short             _height;
   // для экранов с 80 колонками используются специальные алгоритмы
};
```

Screen<hi,80> дает более эффективную реализацию для экранов с 80 столбцами:

Частичная специализация шаблона класса – это шаблон, и ее определение похоже на определение шаблона. Оно начинается с ключевого слова template, за которым следует список параметров, заключенный в угловые скобки. Список параметров здесь отличается от соответствующего списка параметров общего шаблона. Для частичной специализации шаблона Screen есть только один параметр-константа hi, поскольку значение второго

аргумента равно 80, т.е. в данном списке представлены только те параметры, для которых фактические аргументы еще неизвестны.

Имя частичной специализации совпадает с именем того общего шаблона, которому она соответствует, в нашем случае Screen. Однако за ее именем всегда следует список аргументов. В примере выше этот список выглядит как <hi,80>. Поскольку значение аргумента для первого параметра шаблона неизвестно, то на этом месте в списке стоит имя параметра шаблона; вторым же аргументом является значение 80, которым частично специализирован шаблон.

Частичная специализация шаблона класса неявно конкретизируется при использовании в программе. В следующем примере частичная специализация конкретизируется аргументом шаблона 24 вместо hi:

```
Screen<24,80> hp2621;
```

Обратите внимание, что экземпляр Screen<24,80> может быть конкретизирован не только из частично специализированного, но и из общего шаблона. Почему же тогда компилятор остановился именно на частичной специализации? Если для шаблона класса объявлены частичные специализации, компилятор выбирает то определение, которое является наиболее специализированным для заданных аргументов. Если же ни одно из них не подходит, используется общее определение шаблона. Например, при конкретизации экземпляра Screen<40,132> соответствующей аргументам шаблона специализации нет. Наш вариант применяется только для конкретизации типа Screen с 80 колонками.

Определение частичной специализации не связано с определением общего шаблона. У него может быть совершенно другой набор членов, а также собственные определения функций-членов, статических членов и вложенных типов. Содержащиеся в общем шаблоне определения членов никогда не употребляются для конкретизации членов его частичной специализации. Например, для частичной специализации Screen<hi,80>

```
// конструктор для частичной специализации Screen<hi,80>
template <int hi>
Screen<hi,80>::Screen() : _height( hi ), _cursor( 0 ),
                          _screen( hi * 80, bk )
```

должен быть определен свой конструктор:

```
                        { }
```

Если для конкретизации некоторого класса применяется частичная специализация, то определение конструктора из общего шаблона не используется даже тогда, когда определение конструктора Screen<hi,80> отсутствует.

16.11. Разрешение имен в шаблонах классов A

При обсуждении разрешения имен в шаблонах функций (см. раздел 10.9) мы уже говорили о том, что этот процесс выполняется в два шага. Так же разрешаются имена и в определениях шаблонов классов и их членов. Каждый шаг относится к разным видам имен: первый – к тем, которые имеют один и тот же смысл во всех экземплярах шаблона, а второй – к тем, которые потенциально могут иметь разный смысл в разных экземплярах. Рассмотрим несколько примеров, где используется функция-член remove() шаблона класса Queue:

```
// Queue.h:
#include <iostream>
#include <cstdlib>

// определение класса Queue

template <class Type>
Type Queue<Type>::remove() {
   if ( is_empty() ) {
      cerr << "remove() вызвана для пустой очереди\n";
      exit(-1);
   }
   QueueItem<Type> *pt = front;
   front = front->next;
   Type retval = pt->item;
   delete pt;

   cout << "удалено значение: ";
   cout << retval << endl;

   return retval;
}
```

В выражении

```
cout << retval << endl;
```

переменная `retval` имеет тип `Type`, и ее фактический тип неизвестен до конкретизации функции-члена `remove()`. То, какой оператор `operator<<()` будет выбран, зависит от фактического типа `retval`, подставленного вместо `Type`. При разных конкретизациях `remove()` могут вызываться разные `operator<<()`. Поэтому мы говорим, что выбранный оператор вывода *зависит* от параметра шаблона.

Однако для вызова функции `exit()` ситуация иная. Ее фактическим аргументом является литерал, значение которого одинаково при всех конкретизациях `remove()`. Поскольку при обращении к функции не используются аргументы, типы которых зависят от параметра шаблона `Type`, гарантируется, что всегда будет вызываться `exit()`, объявленная в заголовочном файле `cstdlib`. По той же причине в выражении

```
cout << "удалено значение: ";
```

всегда вызывается глобальный оператор

```
ostream& operator<<( ostream &, const char * );
```

Аргумент `"удалено значение: "` – это С-строка символов, и ее тип не зависит от параметра шаблона `Type`. Поэтому в любом конкретизированном экземпляре `remove()`употребление `operator<<()` имеет одинаковый смысл. Один и тот же смысл во всех конкретизациях шаблона имеют те конструкции, которые *не* зависят от параметров шаблона.

Таким образом, два шага разрешения имени в определениях шаблонов классов или их членов состоят в следующем:

- Имена, не зависящие от параметров шаблона, разрешаются во время его определения.

- Имена, зависящие от параметров шаблона, разрешаются во время его конкретизации.

Такой подход удовлетворяет требованиям как разработчика класса, так и его пользователя. Например, разработчикам необходимо управлять процессом разрешения имен. Если шаблон класса входит в состав библиотеки, в которой определены также другие шаблоны и функции, то желательно, чтобы при конкретизации шаблона класса и его членов по возможности применялись именно библиотечные компоненты. Это гарантирует первый шаг разрешения имени. Если использованное в определении шаблона имя не зависит от параметров шаблона, то оно разрешается в результате просмотра всех объявлений, видимых в заголовочном файле, включенном перед определением шаблона.

Разработчик класса должен позаботиться о том, чтобы были видимы объявления всех не зависящих от параметров шаблона имен, употребленных в его определении. Если объявление такого имени не найдено, то определение шаблона считается ошибочным. Если бы перед определением функции-члена `remove()` в шаблоне класса `Queue` не были включены файлы `iostream` и `cstdlib`, то в выражении

```
cout << "удалено значение: ";
```

и при компиляции вызова функции `exit()` были бы обнаружены ошибки.

Второй шаг разрешения имени необходим, если поиск производится среди функций и операторов, зависящих от типа, которым конкретизирован шаблон. Например, если шаблон класса `Queue` конкретизируется типом класса `LongDouble` (см. раздел 16.9), то желательно, чтобы внутри функции-члена `remove()`в следующем выражении

```
cout << retval << endl;
```

```
#include "Queue.h"
#include "ldouble.h"
// содержит:
// class LongDouble { ... };
// ostream& operator<<( ostream &, const LongDouble & );

int main() {
   // конкретизация Queue<LongDouble>
   Queue<LongDouble> *qld = new Queue<LongDouble>;

   // конкретизация Queue<LongDouble>::remove()
   // вызывает оператор вывода для LongDouble
   qld->remove();
   // ...
```

вызывался оператор `operator<<()`, ассоциированный с классом `LongDouble`:

```
}
```

Место в программе, где происходит конкретизация шаблона, называется *точкой конкретизации*. Она определяет, какие объявления принимаются компилятором во внимание для имен, зависящих от параметров шаблона.

Точка конкретизации шаблона всегда находится в области видимости пространства имен и непосредственно предшествует объявлению или определению, которое ссылается на

конкретизированный экземпляр. Точка конкретизации функции-члена или статического члена шаблона класса всегда следует непосредственно за объявлением или определением, которое ссылается на конкретизированный член.

В предыдущем примере точка конкретизации `Queue<LongDouble>` находится перед `main()`, и при разрешении зависящих от параметров имен, которые используются в определении шаблона `Queue`, компилятор просматривает все объявления до этой точки. Аналогично при таком разрешении в определении `remove()` компилятор просматривает все объявления до точки конкретизации, расположенной после `main()`.

Как отмечалось в разделе 16.2, шаблон конкретизируется, если он используется в контексте, требующем полного определения класса. Члены шаблона не конкретизируются автоматически вместе с ним, а лишь тогда, когда сами используются в программе. Поэтому точка конкретизации шаблона класса может не совпадать с точками конкретизации его членов, да и сами члены могут конкретизироваться в разных точках. Чтобы избежать ошибок, объявления имен, упоминаемых в определениях шаблона и его членов, рекомендуется помещать в заголовочные файлы, включая их перед первой конкретизацией шаблона класса или любого из его членов.

16.12. Пространства имен и шаблоны классов

Как и любое определение в глобальной области видимости, определение шаблона класса можно поместить внутрь пространства имен. (Пространства имен рассматривались в разделах 8.5 и 8.6.) Наш шаблон будет скрыт в данном пространстве имен; лишь в этом отличие от ситуации, когда шаблон определен в глобальной области видимости. При употреблении вне пространства имя шаблона следует либо квалифицировать его именем,

```
#include <iostream>
#include <cstdlib>

namespace cplusplus_primer {

   template <class Type>
   class Queue { // ...
   };
   template <class Type>
   Type Queue<Type>::remove()
   {
      // ...
   }
```

либо воспользоваться using-объявлением:

```
}
```

Если имя `Queue` шаблона класса используется вне пространства имен `cplusplus_primer`, то оно должно быть квалифицировано этим именем или введено с помощью using-объявления. Во всех остальных отношениях шаблон `Queue` используется так, как описано выше: конкретизируется, может иметь функции-члены, статические члены, вложенные типы и т.д. Например:

```
int main() {
   using cplusplus_primer Queue;  // using-объявление

   // ссылается на шаблон класса в пространстве имен cplusplus_primer
   Queue<int> *p_qi = new Queue<int>;
   // ...
   p_qi->remove();
}
```

Шаблон cplusplus_primer::Queue<int> конкретизируется, так как использован в выражении new:

```
... = new Queue<int>;
```

p_qi – это указатель на тип класса cplusplus_primer::Queue<int>. Когда он применяется для адресации функции-члена remove(), то речь идет о члене именно этого конкретизированного экземпляра класса.

Объявление шаблона класса в пространстве имен влияет также на объявления специализаций и частичных специализаций шаблона класса и его членов (см. разделы 16.9 и 16.10). Такая специализация должна быть объявлена в том же пространстве имен, где и общий шаблон.

В следующем примере в пространстве имен cplusplus_primer объявляются специализации типа класса Queue<char *> и функции-члена remove() класса

```
#include <iostream>
#include <cstdlib>

namespace cplusplus_primer {

   template <class Type>
   class Queue { ... };

   template <class Type>
   Type Queue<Type>::remove() { ... }

   // объявление специализации
   // для cplusplus_primer::Queue<char *>
   template<> class Queue<char*> { ... };


   // объявление специализации
   // для функции-члена cplusplus_primer::Queue<double>::remove()
   template<> double Queue<double>::remove() { ... }
```

Queue<double>:

```
}
```

Хотя специализации являются членами cplusplus_primer, их определения в этом пространстве отсутствуют. Определить специализацию шаблона можно и вне пространства имен при условии, что определение будет находиться в некотором пространстве, объемлющем cplusplus_primer, и имя специализации будет квалифицировано его именем :

```
namespace cplusplus_primer
{
   // определение Queue и его функций-членов
}

// объявление специализации
// cplusplus_primer::Queue<char*>
template<> class cplusplus_primer::Queue<char*> { ... };

// объявление специализации функции-члена
// cplusplus_primer::Queue<double>::remove()
template<> double cplusplus_primer::Queue<double>::remove()
   { ... }
```

Объявления специализаций класса `cplusplus_primer::Queue<char*>` и функции-члена `remove()` для класса `cplusplus_primer::Queue<double>` находятся в глобальной области видимости. Поскольку такая область содержит пространство имен `cplusplus_primer`, а имена специализаций квалифицированы его именем, то определения специализаций для шаблона `Queue` вполне законны.

16.13. Шаблон класса Array

В этом разделе мы завершим реализацию шаблона класса `Array`, введенного в разделе 2.5 (этот шаблон будет распространен на одиночное наследование в разделе 18.3 и на множественное наследование в разделе 18.6). Так выглядит полный заголовочный файл:

```
#ifndef ARRAY_H
#define ARRAY_H
#include <iostream>

template <class elemType> class Array;
template <class elemType> ostream&
   operator<<( ostream &, Array<elemType> & );

template <class elemType>
class Array {
public:
   explicit Array( int sz = DefaultArraySize )
      { init( 0, sz ); }

   Array( const elemType *ar, int sz )
      { init( ar, sz ); }

   Array( const Array &iA )
      { init( iA._ia, iA._size ); }

   ~Array() { delete[] _ia; }

   Array & operator=( const Array & );
   int size() const { return _size; }

   elemType& operator[]( int ix ) const
      { return _ia[ix]; }

   ostream &print( ostream& os = cout ) const;
   void grow();

   void sort( int,int );
   int find( elemType );
   elemType min();
   elemType max();
private:
   void init( const elemType*, int );
   void swap( int, int );

   static const int DefaultArraySize = 12;

   int _size;
   elemType *_ia;
};

#endif
```

Код, общий для реализации всех трех конструкторов, вынесен в отдельную функцию-член `init()`. Поскольку она не должна напрямую вызываться пользователями шаблона класса `Array`, мы поместили ее в закрытую секцию:

```
template <class elemType>
   void Array<elemType>::init( const elemType *array, int sz )
{
   _size = sz;
   _ia = new elemType[ _size ];

   for ( int ix = 0; ix < _size; ++ix )
      if ( ! array )
         _ia[ ix ] = 0;
      else _ia[ ix ] = array[ ix ];

}
```

Реализация копирующего оператора присваивания не вызывает затруднений. Как

```
template <class elemType> Array<elemType>&
   Array<elemType>::operator=( const Array<elemType> &iA )
{
   if ( this != &iA ) {
      delete[] _ia;
      init( iA._ia, iA._size );
   }
   return *this;
}
```

отмечалось в разделе 14.7, в код включена защита от копирования объекта в самого себя:

Функция-член `print()` отвечает за вывод объекта того типа, которым конкретизирован шаблон `Array`. Возможно, реализация несколько сложнее, чем необходимо, зато данные аккуратно размещаются на странице. Если экземпляр конкретизированного класса `Array<int>` содержит элементы 3, 5, 8, 13 и 21, то выведены они будут так:

```
(5) < 3, 5, 8, 13, 21 >
```

Оператор потокового вывода просто вызывает `print()`. Ниже приведена реализация обеих функций:

```
template <class elemType> ostream&
   operator<<( ostream &os, Array<elemType> &ar )
{
   return ar.print( os );
}

template <class elemType>
   ostream & Array<elemType>::print( ostream &os ) const
{
   const int lineLength = 12;

   os << "( " << _size << " )< ";
   for ( int ix = 0; ix < _size; ++ix )
   {
      if ( ix % lineLength == 0 && ix )
         os << "\n\t";
      os << _ia[ ix ];

      // не выводить запятую за последним элементом в строке,
      // а также за последним элементом массива
      if ( ix % lineLength != lineLength-1 && ix != _size-1 )
         os << ", ";
   }

   os << " >\n";
   return os;
}
```

Вывод значения элемента массива в функции `print()` осуществляет такая инструкция:

```
os << _ia[ ix ];
```

Для ее правильной работы должно выполняться требование к типам, которыми конкретизируется шаблон `Array`: такой тип должен быть встроенным либо иметь собственный оператор вывода. В противном случае любая попытка распечатать содержимое класса `Array` приведет к ошибке компиляции в том месте, где используется несуществующий оператор.

Функция-член `grow()` увеличивает размер объекта класса `Array`. В нашем примере – в полтора раза:

```
template <class elemType>
    void Array<elemType>::grow()
{
    elemType *oldia = _ia;
    int oldSize = _size;

    _size = oldSize + oldSize/2 + 1;
    _ia   = new elemType[_size];

    int ix;
    for ( ix = 0; ix < oldSize; ++ix )
          _ia[ix] = oldia[ix];

    for ( ; ix < _size; ++ix )
          _ia[ix] = elemType();

    delete[] oldia;

}
```

Функции-члены `find()`, `min()` и `max()` осуществляют последовательный поиск во внутреннем массиве `_ia`. Если бы массив был отсортирован, то, конечно, их можно было

```
template <class elemType>
    elemType Array<elemType>::min( )
{
    assert( _ia != 0 );
    elemType min_val = _ia[0];
    for ( int ix = 1; ix < _size; ++ix )
       if ( _ia[ix] < min_val )
          min_val = _ia[ix];

    return min_val;
}

template <class elemType>
    elemType Array<elemType>::max()
{
    assert( _ia != 0 );
    elemType max_val = _ia[0];

    for ( int ix = 1; ix < _size; ++ix )
       if ( max_val < _ia[ix] )
          max_val = _ia[ix];

    return max_val;
}

template <class elemType>
    int Array<elemType>::find( elemType val )
{
    for ( int ix = 0; ix < _size; ++ix )
       if ( val == _ia[ix] )
          return ix;

    return -1;
```

бы реализовать гораздо эффективнее.

```
}
```

В шаблоне класса Array есть функция-член sort(), реализованная с помощью алгоритма быстрой сортировки. Она очень похожа на шаблон функции, представленный в разделе 10.11. Функция-член swap() – вспомогательная утилита для sort(); она не

```
template <class elemType>
    void Array<elemType>::swap( int i, int j )
{
     elemType tmp = _ia[i];
     _ia[i] = _ia[j];
     _ia[j] = tmp;
}

template <class elemType>
    void Array<elemType>::sort( int low, int high )
{
    if ( low >= high ) return;
    int lo = low;
    int hi = high + 1;
    elemType elem = _ia[low];

    for ( ;; ) {
         while ( _ia[++lo] < elem ) ;
         while ( _ia[--hi] > elem ) ;
         if ( lo < hi )
              swap( lo,hi );
         else break;
    }

    swap( low, hi );
    sort( low, hi-1 );
    sort( hi+1, high );
```

является частью открытого интерфейса шаблона и потому помещена в закрытую секцию:

```
}
```

То, что код реализован, разумеется, не означает, что он работоспособен. try_array() – это шаблон функции, предназначенный для тестирования реализации шаблона Array:

```
#include "Array.h"

template <class elemType>
    void try_array( Array<elemType> &iA )
{
    cout << "try_array: начальные значения массива\n";
    cout << iA << endl;

    elemType find_val = iA [ iA.size()-1 ];
    iA[ iA.size()-1 ] = iA.min();

    int mid = iA.size()/2;
    iA[0] = iA.max();
    iA[mid] = iA[0];
    cout << "try_array: после присваиваний\n";
    cout << iA << endl;

    Array<elemType> iA2 = iA;
    iA2[mid/2] = iA2[mid];
    cout << "try_array: почленная инициализация\n";
    cout << iA << endl;

    iA = iA2;
    cout << "try_array: после почленного копирования\n";
    cout << iA << endl;

    iA.grow();
    cout << "try_array: после вызова grow\n";
    cout << iA << endl;
    int index = iA.find( find_val );
    cout << "искомое значение: " << find_val;
    cout << "\tвозвращенный индекс: " << index << endl;

    elemType value = iA[index];
    cout << "значение элемента с этим индексом: ";
    cout << value << endl;

}
```

Рассмотрим шаблон функции `try_array()`. На первом шаге печатается исходный объект `Array`, что подтверждает успешную конкретизацию оператора вывода шаблона, а заодно дает начальную картину, с которой можно будет сверяться при последующих модификациях. В переменной `find_val` хранится значение, которое мы впоследствии передадим `find()`. Если бы `try_array()` была обычной функцией, роль такого значения сыграла бы константа. Но поскольку никакая константа не может обслужить все типы, которыми допустимо конкретизировать шаблон, то приходится выбирать другой путь. Далее одним элементам `Array` случайным образом присваиваются значения других элементов, чтобы протестировать `min()`, `max()`, `size()` и, конечно, оператор взятия индекса.

Затем объект `iA2` почленно инициализируется объектом `iA`, что приводит к вызову копирующего конструктора. После этого тестируется оператор взятия индекса с объектом `ia2`: производится присваивание элементу с индексом `mid/2`. (Эти две строки представляют интерес в случае, когда `iA` – производный подтип `Array`, а оператор взятия индекса объявлен виртуальной функцией. Мы вернемся к этому в главе 18 при обсуждении наследования.) Далее в `iA` почленно копируется модифицированный объект `iA2`, что приводит к вызову копирующего оператора присваивания класса `Array`. Затем проверяются функции-члены `grow()` и `find()`. Напомним, что `find()` возвращает значение –1, если искомый элемент не найден. Попытка выбрать из “массива” `Array`

элемент с индексом –1 приведет к выходу за левую границу. (В главе 18 для перехвата этой ошибки мы построим производный от Array класс, который будет проверять выход за границы массива.)

Убедиться, что наша реализация шаблона работает для различных типов данных, например целых чисел, чисел с плавающей точкой и строк, поможет программа main(),

```
#include "Array.C"
#include "try_array.C"
#include <string>

int main()
{
    static int ia[] = { 12,7,14,9,128,17,6,3,27,5 };
    static double da[] = { 12.3,7.9,14.6,9.8,128.0 };
    static string sa[] = {
        "Eeyore", "Pooh", "Tigger",
        "Piglet", "Owl", "Gopher", "Heffalump"
    };
    Array<int>    iA( ia, sizeof(ia)/sizeof(int) );
    Array<double> dA( da, sizeof(da)/sizeof(double) );
    Array<string> sA( sa, sizeof(sa)/sizeof(string) );

    cout << "template Array<int> class\n" << endl;
    try_array(iA);

    cout << "template Array<double> class\n" << endl;
    try_array(dA);

    cout << "template Array<string> class\n" << endl;
    try_array(sA);

    return 0;
```

которая вызывает try_array() с каждым из указанных типов:

```
}
```

Вот что программа выводит при конкретизации шаблона Array типом double:

```
try_array: начальные значения массива
( 5 )< 12.3, 7.9, 14.6, 9.8, 128 >

try_array: после присваиваний
( 5 )< 14.6, 7.9, 14.6, 9.8, 7.9 >

try_array: почленная инициализация
( 5 )< 14.6, 7.9, 14.6, 9.8, 7.9 >

try_array: после почленного копирования
( 5 )< 14.6, 14.6, 14.6, 9.8, 7.9 >

try_array: после вызова grow
( 8 )< 14.6, 14.6, 14.6, 9.8, 7.9, 0, 0, 0 >

искомое значение: 128      возвращенный индекс: -1
значение элемента с этим индексом: 3.35965e-322
```

Выход индекса за границу массива приводит к тому, что последнее напечатанное программой значение неверно. Конкретизация шаблона Array типом string заканчивается крахом программы:

```
template Array<string> class

try_array: начальные значения массива
( 7 )< Eeyore, Pooh, Tigger, Piglet, Owl, Gopher, Heffalump >

try_array: после присваиваний
( 7 )< Tigger, Pooh, Tigger, Tigger, Owl, Gopher, Eeyore >

try_array: почленная инициализация
( 7 )< Tigger, Pooh, Tigger, Tigger, Owl, Gopher, Eeyore >

try_array: после почленного копирования
( 7 )< Tigger, Tigger, Tigger, Tigger, Owl, Gopher, Eeyore >

try_array: после вызова grow
( 11 )< Tigger, Tigger, Tigger, Tigger, Owl, Gopher, Eeyore, <пусто>, <пусто>,
<пусто>, <пусто>, <пусто> >

искомое значение: Heffalump          возвращенный индекс: -1
Memory fault (coredump)
```

Упражнение 16.11

Измените шаблон класса `Array`, убрав из него функции-члены `sort()`, `find()`, `max()`, `min()` и `swap()`, и модифицируйте шаблон `try_array()` так, чтобы она вместо них пользовалась обобщенными алгоритмами (см. главу 12).

Часть V

Объектно-ориентированное программирование

Объектно-ориентированное программирование расширяет объектное программирование, вводя отношения тип-подтип с помощью механизма, именуемого *наследованием*. Вместо того чтобы заново реализовывать общие свойства, класс наследует данные-члены и функции-члены родительского класса. В языке C++ наследование осуществляется посредством так называемого *порождения производных классов*. Класс, свойства которого наследуются, называется *базовым*, а новый класс – *производным*. Все множество базовых и производных классов образует *иерархию* наследования.

Например, в трехмерной компьютерной графике классы `OrthographicCamera` и `PerspectiveCamera` обычно являются производными от базового `Camera`. Множество операций и данных, общее для всех камер, определено в абстрактном классе `Camera`. Каждый производный от него класс реализует лишь отличия от абстрактной камеры, предоставляя альтернативный код для унаследованных функций-членов либо вводя дополнительные члены.

Если базовый и производный классы имеют общий открытый интерфейс, то производный называется *подтипом* базового. Так, `PerspectiveCamera` является подтипом класса `Camera`. В C++ существует специальное отношение между типом и подтипом, позволяющее указателю или ссылке на базовый класс адресовать любой из производных от него подтипов без вмешательства программиста. (Такая возможность манипулировать несколькими типами с помощью указателя или ссылки на базовый класс называется *полиморфизмом*.) Если дана функция:

```
void lookAt( const Camera *pCamera );
```

то мы реализуем `lookAt()`, программируя интерфейс базового класса `Camera` и не заботясь о том, на что указывает `pCamera`: на объект класса `PerspectiveCamera`, на объект класса `OrthographicCamera` или на объект, описывающий еще какой-то вид камеры, который мы пока не определили.

При каждом вызове `lookAt()` ей передается адрес объекта, принадлежащего к одному из подтипов `Camera`. Компилятор автоматически преобразует его в указатель на подходящий базовый класс:

```
// правильно: автоматически преобразуется в Camera*
OrthographicCamera ocam;
lookAt( &ocam );

// ...

// правильно: автоматически преобразуется в Camera*
PerspectiveCamera *pcam = new PerspectiveCamera;
lookAt( pcam );
```

Наша реализация `lookAt()` не зависит от набора подтипов класса `Camera`, реально существующих в приложении. Если впоследствии потребуется добавить новый подтип или исключить существующий, то изменять реализацию `lookAt()` не придется.

Полиморфизм подтипов позволяет написать ядро приложения так, что оно не будет зависеть от конкретных типов, которыми мы манипулируем. Мы программируем открытый интерфейс базового класса придуманной нами абстракции, пользуясь только ссылками и указателями на него. При работе программы будет определен фактический тип адресуемого объекта и вызвана подходящая реализация открытого интерфейса.

Нахождение (или разрешение) нужной функции во время выполнения называется *динамическим связыванием* (dynamic binding) (по умолчанию функции разрешаются *статически* во время компиляции). В C++ динамическое связывание поддерживается с помощью механизма *виртуальных функций* класса. Полиморфизм подтипов и динамическое связывание формируют основу объектно-ориентированного программирования, которому посвящены следующие главы.

В главе 17 рассматриваются имеющиеся в C++ средства поддержки объектно-ориентированного программирования и изучается влияние наследование на такие механизмы, как конструкторы, деструкторы, почленная инициализация и присваивание; для примера разрабатывается иерархия классов `Query`, поддерживающая систему текстового поиска, введенную в главе 6.

Темой главы 18 является изучение более сложных иерархий, возможных за счет использования множественного и виртуального наследования. С его помощью мы развернем шаблон класса из главы 16 в трехуровневую иерархию.

В главе 19 обсуждается идентификация типов во время выполнения (RTTI), а также изучается вопрос о влиянии наследования на разрешение перегруженных функций. Здесь мы снова обратимся к средствам обработки исключений, чтобы разобраться в иерархии классов исключений, которую предлагает стандартная библиотека. Мы покажем также, как написать собственные такие классы.

Глава 20 посвящена углубленному рассмотрению библиотеки потокового ввода/вывода `iostream`. Эта библиотека представляет собой иерархию классов, поддерживающую как виртуальное, так и множественное наследование.

17

17. Наследование и подтипизация классов

В главе 6 для иллюстрации обсуждения абстрактных контейнерных типов мы частично реализовали систему текстового поиска и инкапсулировали ее в класс `TextQuery`. Однако мы не написали к ней никакой вызывающей программы, отложив реализацию поддержки формулирования запросов со стороны пользователя до рассмотрения объектно-ориентированного программирования. В этой главе язык запросов будет реализован в виде иерархии классов `Query` с одиночным наследованием. Кроме того, мы модифицируем и расширим класс `TextQuery` из главы 6 для получения полностью интегрированной системы текстового поиска.

Программа для запуска нашей системы текстового поиска будет выглядеть следующим образом:

```
#include "TextQuery.h"

int main()
{
   TextQuery tq;

   tq.build_up_text();
   tq.query_text();
}
```

build_text_map() – это слегка видоизмененная функция-член doit() из главы 6. Ее основная задача – построить отображение для хранения позиций всех значимых слов текста. (Если помните, мы не храним семантически нейтральные слова типа союзов if, and, but и т.д. Кроме того, мы заменяем заглавные буквы на строчные и устраняем суффиксы, обозначающие множественное число: например, testifies преобразуется в testify, а marches в march.) С каждым словом ассоциируется вектор позиций, в котором хранятся номера строки и колонки каждого вхождения слова в текст.

query_text() принимает запросы пользователя и преобразует их во внутреннюю форму на основе иерархии классов Query с одиночным наследованием и динамическим связыванием. Внутреннее представление запроса применяется к отображению слов на вектор позиций, построенному в build_text_map(). Ответом на запрос будет множество строк текстового файла, удовлетворяющих заданному критерию:

```
Enter a query - please separate each item by a space.
Terminate query (or session) with a dot( . ).

==> fiery && ( bird || shyly )

         fiery ( 1 ) lines match
         bird ( 1 ) lines match
         shyly ( 1 ) lines match
          ( bird || shyly ) ( 2 ) lines match
         fiery && ( bird || shyly ) ( 1 ) lines match

Requested query: fiery && ( bird || shyly )

( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her.
```

В нашей системе мы выбрали следующий язык запросов:

- одиночное слово, например Alice или untamed. Выводятся все строки, в которых оно встречается, причем каждой строке предшествует ее номер, заключенный в скобки. (Строки печатаются в порядке возрастания номеров). Например:

```
==> daddy

    daddy ( 3 ) lines match

Requested query: daddy

( 1 ) Alice Emma has long flowing red hair. Her Daddy says
( 4 ) magical but untamed. "Daddy, shush, there is no such thing,"
( 6 ) Shyly, she asks, "I mean, Daddy, is there?"
```

- запрос “НЕ”, формулируемый с помощью оператора `!`. Выводятся все строки, где не встречается указанное слово. Например, так формулируется отрицание запроса 1:

```
==> ! daddy

    daddy ( 3 ) lines match
    ! daddy ( 3 ) lines match

Requested query: ! daddy

( 2 ) when the wind blows through her hair, it looks almost alive,
( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
( 5 ) she tells him, at the same time wanting him to tell her more.
```

запрос “ИЛИ”, формулируемый с помощью оператора `||`. Выводятся все строки, в которых встречается хотя бы одно из двух указанных слов:

```
==> fiery || untamed

        fiery ( 1 ) lines match
        untamed ( 1 ) lines match
        fiery || untamed ( 2 ) lines match

Requested query: fiery || untamed

( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
( 4 ) magical but untamed. "Daddy, shush, there is no such thing,"
```

запрос “И”, формулируемый с помощью оператора `&&`. Выводятся все строки, где оба указанных слова встречаются, причем располагаются рядом. Сюда входит и случай, когда одно слово является последним в строке, а другое – первым в следующей:

```
==> untamed && Daddy

        untamed ( 1 ) lines match
        daddy ( 3 ) lines match
        untamed && daddy ( 1 ) lines match

Requested query: untamed && daddy

( 4 ) magical but untamed. "Daddy, shush, there is no such thing,"
```

Эти элементы можно комбинировать:

```
fiery && bird || shyly
```

Однако обработка производится слева направо, и все элементы имеют одинаковые приоритеты. Поэтому наш составной запрос интерпретируется как `fiery bird` ИЛИ `shyly`, а не как `fiery bird` ИЛИ `fiery shyly`:

```
==> fiery && bird || shyly

        fiery ( 1 ) lines match
        bird ( 1 ) lines match
        fiery && bird ( 1 ) lines match
        shyly ( 1 ) lines match
        fiery && bird || shyly ( 2 ) lines match
```

```
Requested query: fiery && bird || shyly

( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
( 6 ) Shyly, she asks, "I mean, Daddy, is there?"
```

Чтобы можно было группировать части запроса, наша система должна поддерживать скобки. Например:

```
fiery && (bird || shyly)
```

выдает все вхождения `fiery bird` или `fiery shyly`[1]. Результат исполнения этого запроса приведен в начале данного раздела. Кроме того, система не должна многократно отображать одну и ту же строку.

Примечание [O.A.5]: Нумерация сносок сбита.

17.1. Определение иерархии классов

В этой главе мы построим иерархию классов для представления запроса пользователя.

```
NameQuery    // Shakespeare
NotQuery     // ! Shakespeare
OrQuery      // Shakespeare || Marlowe
```

Сначала реализуем каждую операцию в виде отдельного класса:

```
AndQuery     // William && Shakespeare
```

В каждом классе определим функцию-член `eval()`, которая выполняет соответствующую операцию. К примеру, для `NameQuery` она возвращает вектор позиций, содержащий координаты (номера строки и колонки) начала каждого вхождения слова (см. раздел 6.8); для `OrQuery` строит объединение векторов позиций обоих своих операндов и т.д.

Таким образом, запрос

```
untamed || fiery
```

состоит из объекта класса `OrQuery`, который содержит два объекта `NameQuery` в качестве операндов. Для простых запросов этого достаточно, но при обработке составных запросов типа

```
Alice || Emma && Weeks
```

---

1 Напомним, что для упрощения реализации необходимо, чтобы между любыми двумя словами, включая скобки и операторы запроса, был пробел. В реальной системе такое требование вряд ли разумно, но мы полагаем, что для вводного курсе, каковым является наша книга, это вполне приемлемо.

возникает проблема. Данный запрос состоит из двух подзапросов: объекта OrQuery, содержащего объекты NameQuery для представления слов Alice и Emma, и объекта

```
AndQuery
    OrQuery
        NameQuery ("Alice")
        NameQuery ("Emma")
```

AndQuery. Правым операндом AndQuery является объект NameQuery для слова Weeks.

```
    NameQuery ("Weeks")
```

Но левый операнд – это объект OrQuery, предшествующий оператору &&. На его месте мог бы быть объект NotQuery или другой объект AndQuery. Как же следует представить операнд, если он может принадлежать к типу любого из четырех классов? Эта проблема имеет две стороны:

- необходимо уметь объявлять тип операнда в классах OrQuery, AndQuery и NotQuery так, чтобы с его помощью можно было представить тип любого из четырех классов запросов;
- какое бы решение мы ни выбрали в предыдущем случае, мы должны иметь возможность вызывать соответствующий классу каждого операнда вариант функции-члена eval().

Решение, не согласующееся с объектной ориентированностью, состоит в том, чтобы определить тип операнда как объединение и включить *дискриминант*, показывающий

```
// не объектно-ориентированное решение
union op_type {
   // объединение не может содержать объекты классов с
   // ассоциированными конструкторами
   NotQuery *nq;
   OrQuery  *oq;
   AndQuery *aq;
   string   *word;
};

enum opTypes {
   Not_query=1, O_query, And_query, Name_query
};

class AndQuery {
public:
   // ...
private:
   /*
    * opTypes хранит информацию о фактических типах операндов запроса
    * op_type - это сами операнды
    */

    op_type _lop, _rop;
    opTypes _lop_type, _rop_type;
```

текущий тип операнда:

```
};
```

Хранить указатели на объекты можно и с помощью типа void*:

```
class AndQuery {
public:
   // ...
private:
    void * _lop, _rop;
    opTypes _lop_type, _rop_type;
};
```

Нам все равно нужен дискриминант, поскольку напрямую использовать объект, адресуемый указателем типа `void*`, нельзя, равно как невозможно определить тип такого объекта по указателю. (Мы не рекомендуем применять описанное решение в C++, хотя в языке C это весьма распространенный подход.)

Основной недостаток рассмотренных решений состоит в том, что ответственность за определение типа возлагается на программиста. Например, в случае решения, основанного на `void*`-указателях, операцию `eval()` для объекта `AndQuery` можно

```
void
AndQuery::
eval()
{
   // не объектно-ориентированный подход
   // ответственность за разрешение типа ложится на программиста

   // определить фактический тип левого операнда
   switch( _lop_type ) {
      case And_query:
           AndQuery *paq = static_cast<AndQuery*>(_lop);
           paq->eval();
           break;
      case Or_query:
           OrQuery *pqq = static_cast<OrQuery*>(_lop);
           poq->eval();
           break;
      case Not_query:
           NotQuery *pnotq = static_cast<NotQuery*>(_lop);
           pnotq->eval();
           break;
      case Name_query:
           AndQuery *pnmq = static_cast<NameQuery*>(_lop);
           pnmq->eval();
           break;
   }

   // то же для правого операнда
}
```

реализовать так:

В результате явного управления разрешением типов увеличивается размер и сложность кода и добавление нового типа или исключение существующего при сохранении работоспособности программы затрудняется.

Объектно-ориентированное программирование предлагает альтернативное решение, в котором работа по разрешению типов перекладывается с программиста на компилятор. Например, так выглядит код операции `eval()` для класса `AndQuery` в случае применения объектно-ориентированного подхода (`eval()` объявлена виртуальной):

```
// объектно-ориентированное решение
// ответственность за разрешение типов перекладывается на компилятор

// примечание: теперь _lop и _rop - объекты типа класса
// их определения будут приведены ниже

void
AndQuery::
eval()
{
   _lop->eval();
   _rop->eval();
}
```

Если потребуется добавить или исключить какие-либо типы, эту часть программы не придется ни переписывать, ни перекомпилировать.

17.1.1. Объектно-ориентированное проектирование

Из чего складывается объектно-ориентированное проектирование четырех рассмотренных выше видов запросов? Как решаются проблемы их внутреннего представления?

С помощью *наследования* можно определить взаимосвязи между независимыми классами запросов. Для этого мы вводим в рассмотрение абстрактный класс `Query`, который будет служить для них *базовым* (соответственно сами эти классы будут считаться *производными*). Абстрактный класс можно представить себе как неполный, который становится более или менее завершенным, когда из него порождаются производные классы, – в нашем случае `AndQuery`, `OrQuery`, `NotQuery` и `NameQuery`.

В нашем абстрактном классе `Query` определены данные и функции-члены, общие для всех четырех типов запроса. При порождении из `Query` производного класса, скажем `AndQuery`, мы выделяем уникальные характеристики каждого вида запроса. К примеру, `NameQuery` – это специальный вид `Query`, в котором операндом всегда является строка. Мы будем называть `NameQuery` *производным* и говорить, что `Query` является его *базовым классом*. (То же самое относится и к классам, представляющим другие типы запросов.) Производный класс наследует данные и функции-члены базового и может обращаться к ним непосредственно, как к собственным членам.

Основное преимущество иерархии наследования в том, что мы программируем открытый интерфейс абстрактного базового класса, а не отдельных производных от него специализированных типов, что позволяет защитить наш код от последующих изменений иерархии. Например, мы определяем `eval()` как открытую виртуальную функцию абстрактного базового класса `Query`. Пользовательский код, записанный в виде:

```
_rop->eval();
```

экранирован от любых изменений в языке запросов. Это не только позволяет добавлять, модифицировать и удалять типы, не изменяя программы пользователя, но и освобождает автора нового вида запроса от необходимости заново реализовывать поведение или действия, общие для всех типов в иерархии. Такая гибкость достигается за счет двух характеристик механизма наследования: *полиморфизма* и *динамического связывания*.

Когда мы говорим о полиморфизме в языке C++, то имеем в виду главным образом способность указателя или ссылки на базовый класс адресовать любой из производных от

```
// pquery может адресовать любой из классов, производных от Query
void eval( const Query *pquery )
{
   pquery->eval();
}
```

него. Если определить обычную функцию `eval()` следующим образом:

```
int main()
{
   AndQuery aq;
   NotQuery notq;
   OrQuery *oq = new OrQuery;
   NameQuery nq( "Botticelli" );

   // правильно: любой производный от Query класс
   // компилятор автоматически преобразует в базовый класс
   eval( &aq );
   eval( &notq );
   eval( oq );
   eval( &nq );
}
```

то мы вправе вызывать ее, передавая адрес объекта любого из четырех типов запросов:

В то же время попытка передать `eval()` адрес объекта класса, не являющегося

```
int main()
{
   string name( "Scooby-Doo" );

   // ошибка: тип string не является производным от Query
   eval( &name );
}
```

производным от `Query`, вызовет ошибку компиляции:

Внутри `eval()` выполнение инструкции вида

```
pquery->eval();
```

должно вызывать нужную виртуальную функцию-член `eval()` в зависимости от фактического класса объекта, адресуемого указателем `pquery`. В примере выше `pquery` последовательно адресует объекты `AndQuery`, `NotQuery`, `OrQuery` и `NameQuery`. В каждой точке вызова определяется фактический тип класса объекта и вызывается подходящий экземпляр `eval()`.

Механизм, с помощью которого это достигается, называется *динамическим связыванием*. (Мы вернемся к проектированию и использованию виртуальных функций в разделе 17.5.)

В объектно-ориентированной парадигме программист манипулирует неизвестным экземпляром, принадлежащим к одному из ограниченного, но потенциально бесконечного множества различных типов. (Ограничено оно иерархией наследования. Теоретически, однако, ни на глубину, ни на ширину такой иерархии не накладывается никаких ограничений.) В C++ это достигается путем манипулирования объектами исключительно через указатели и ссылки на базовый класс. В объектной (не объектно-ориентированной) парадигме программист работает с экземпляром фиксированного типа, который полностью определен на этапе компиляции.

Хотя для полиморфной манипуляции объектом требуется, чтобы доступ к нему осуществлялся с помощью указателя или ссылки, сам по себе факт их использования не

```
// полиморфизма нет
int *pi;

// нет поддержанного языком полиморфизма
void *pvi;

// pquery может адресовать объект любого производного от Query класса
```

обязательно приводит к полиморфизму. Рассмотрим такие объявления:

```
Query *pquery;
```

В C++ полиморфизм существует только в пределах отдельных иерархий классов. Указатели типа `void*` можно назвать полиморфными, но в языке их поддержка не предусмотрена. Такими указателями программист должен управлять самостоятельно, с помощью явных приведений типов и той или иной формы дискриминанта, показывающего, объект какого типа в данный момент адресуется. (Можно сказать, что это “второсортные” полиморфные объекты.)

Язык C++ обеспечивает поддержку полиморфизма следующими способами:

- путем неявного преобразования указателя или ссылки на производный класс к указателю или ссылке на открытый базовый:

```
Query *pquery = new NameQuery( "Class" );
```

- через механизм виртуальных функций:

```
pquery->eval();
```

- с помощью операторов `dynamic_cast` и `typeid` (они подробно обсуждаются в

```
if ( NameQuery *pnq =
```

разделе 19.1):

```
  dynamic_cast< NameQuery* >( pquery )) ...
```

Проблему представления запроса мы решим, определив каждый операнд в классах `AndQuery`, `NotQuery` и `OrQuery` как указатель на тип `Query*`. Например:

```
class AndQuery {
public:
   // ...
private:
   Query *_lop;
   Query *_rop;
};
```

Теперь оба операнда могут адресовать объект любого класса, производного от абстрактного базового класса `Query`, без учета того, определен он уже сейчас или появится в будущем. Благодаря механизму виртуальных функций, вычисление операнда, происходящее во время выполнения программы, не зависит от фактического типа:

```
_rop->eval();
```

На рис. 17.1 показана иерархия наследования, состоящая из абстрактного класса `Query` и четырех производных от него классов. Как этот рисунок транслируется в код программы на C++?

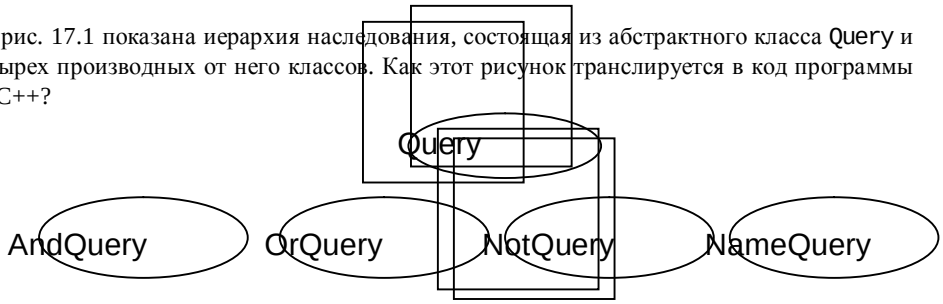


Рис. 17.1. Иерархия классов Query

В разделе 2.4 мы рассматривали реализацию иерархии классов `IntArray`. Синтаксическая структура определения иерархии, изображенной на рис. 17.1,

```
class Query { ... };
class AndQuery   : public Query { ... };
class OrQuery    : public Query { ... };
class NotQuery   : public Query { ... };
```

аналогична:

```
class NameQuery  : public Query { ... };
```

Наследование задается с помощью *списка базовых классов*. В случае одиночного наследования этот список имеет вид:

```
: уровень-доступа базовый-класс
```

где `уровень-доступа` – это одно из ключевых слов `public`, `protected`, `private` (смысл защищенного и закрытого наследования мы обсудим в разделе 18.3), а `базовый-класс` – имя ранее определенного класса. Например, `Query` является открытым базовым классом для любого из четырех классов запросов.

Класс, встречающийся в списке базовых, должен быть предварительно определен. Следующего опережающего объявления `Query` недостаточно для того, чтобы он мог выступать в роли базового:

```
// ошибка: Query должен быть определен
class Query;

class NameQuery : piblic Query { ... };
```

Опережающее объявление производного класса должно включать только его имя, но не список базовых классов. Поэтому следующее опережающее объявление класса NameQuery

```
// ошибка: опережающее объявление не должно
// включать списка базовых классов
```

приводит к ошибке компиляции:

```
class NameQuery : public Query;
```

```
// опережающее объявление как производного,
// так и обычного класса содержит только имя класса
class Query;
```

Правильный вариант в данном случае выглядит так:

```
class NameQuery;
```

Главное различие между базовыми классами Query и IntArray (см. раздел 2.4) состоит в том, что Query не представляет никакого реального объекта в нашем приложении. Пользователи класса IntArray вполне могут определять и использовать объекты этого типа непосредственно. Что же касается Query, то разрешается определять лишь указатели и ссылки на него, используя их для косвенного манипулирования объектами производных классов. О Query говорят, что это *абстрактный базовый класс*. В противоположность этому IntArray является *конкретным базовым классом*. Преобладающей формой в объектно-ориентированном проектировании является определение абстрактного базового класса типа Query и одиночное открытое наследование ему.

Упражнение 17.1

Библиотека может выдавать на руки предметы, для каждого из которых определены

```
книга                        аудио-книга
аудиокассета                 детская кукла
видеокассета                 видеоигра для приставки SEGA
книга с подневной оплатой    видеоигра для приставки SONY
```

специальные правила выдачи и возврата. Организуйте их в иерархию наследования:

```
книга на компакт-диске       видеоигра для приставки Nintendo
```

Упражнение 17.2

Выберите или придумайте собственную абстракцию, содержащую семейство типов. Организуйте типы в иерархию наследования:

(a) Форматы графических файлов (gif, tiff, jpeg, bmp и т.д.)

(b) Геометрические примитивы (прямоугольник, круг, сфера, конус и т.д.)

(c) Типы языка C++ (класс, функция, функция-член и т.д.)

17.2. Идентификация членов иерархии

В разделе 2.4 мы уже упоминали о том, что в объектном проектировании обычно есть один разработчик, который конструирует и реализует класс, и много пользователей, применяющих предоставленный открытый интерфейс. Это разделение ответственности отразилось в концепции открытого и закрытого доступа к членам класса.

Когда используется наследование, у класса оказывается множество разработчиков. Во-первых, тот, кто предоставил реализацию базового класса (и, возможно, некоторых производных от него), а во-вторых, те, кто разрабатывал производные классы на различных уровнях иерархии. Этот род деятельности тоже относится к проектированию. Разработчик подтипа часто (хотя и не всегда) должен иметь доступ к реализации базового класса. Чтобы разрешить такой вид доступа, но все же предотвратить неограниченный доступ к деталям реализации класса, вводится дополнительный уровень доступа – `protected` (защищенный). Данные и функции-члены, помещенные в секцию `protected` некоторого класса, остаются недоступными вызывающей программе, но обращение к ним из производных классов разрешено. (Все находящееся в секции `private` базового класса доступно только ему, но не производным.)

Критерии помещения того или иного члена в секцию `public` одинаковы как для объектного, так и для объектно-ориентированного проектирования. Меняется только точка зрения на то, следует ли объявлять член закрытым или защищенным. Член базового класса объявляется закрытым, если мы не хотим, чтобы производные классы имели к нему прямой доступ; и защищенным, если его семантика такова, что для эффективной реализации производного класса может потребоваться прямой доступ к нему. При проектировании класса, который предполагается использовать в качестве базового, надо также принимать во внимание особенности функций, зависящих от типа, – виртуальных функций в иерархии классов.

На следующем шаге проектирования иерархии классов `Query` следует ответить на такие вопросы:

(a) Какие операции следует предоставить в открытом интерфейсе иерархии классов `Query`?

(b) Какие из них следует объявить виртуальными?

(c) Какие дополнительные операции могут потребоваться производным классам?

(d) Какие данные-члены следует объявить в нашем абстрактном базовом классе `Query`?

(e) Какие данные-члены могут потребоваться производным классам?

К сожалению, однозначно ответить на эти вопросы невозможно. Как мы увидим, процесс объектно-ориентированного проектирования по своей природе итеративен, эволюционирующая иерархия классов требует и добавлений, и модификаций. В оставшейся части этого раздела мы будем постепенно уточнять иерархию классов `Query`.

17.2.1. Определение базового класса

Члены `Query` представляют:

- множество операций, поддерживаемых всеми производными от него классами запросов. Сюда входят как виртуальные операции, переопределяемые в производных классах, так и невиртуальные, разделяемые всеми производными классами (мы приведем примеры тех и других);
- множество данных-членов, общих для всех производных классов. Если вынести такие члены в абстрактный базовый класс `Query`, мы сможем обращаться к ним вне зависимости от того, с объектом какого производного класса мы работаем.

Если имеется запрос вида:

```
fiery || untamed
```

то двумя основными операциями для него будут: нахождение строк текста, удовлетворяющих условиям запроса, и представление найденных строк пользователю. Назовем эти операции соответственно `eval()` и `display()`.

Алгоритм работы `eval()` свой для каждого производного класса, поэтому эту функцию следует объявить виртуальной в определении `Query`. Всякий производный класс должен предоставить собственную реализацию для нее. Сам же `Query` лишь включает ее в свой открытый интерфейс.

Алгоритм работы функции `display()`, выводящей найденные строки текста, не зависит от типа производного класса. Нам необходимо лишь иметь доступ к представлению самого текста и списку строк, удовлетворяющих запросу. Вместо того чтобы дублировать реализацию алгоритма и необходимые для него данные в каждом производном классе, определим единственный наследуемый экземпляр в `Query`.

Такое проектное решение позволит нам вызывать любую операцию, не зная фактического

```
void
doit( Query *pq )
{
   // виртуальный вызов
   pq->eval();

   // статический вызов Query::display()
   pq->display();
}
```

типа объекта, которым мы манипулируем:

Как следует представить найденные строки текста? Каждому упомянутому в запросе слову будет соответствовать вектор позиций, построенный во время поиска. Позиция – это пара (строка, колонка), в которой каждый член – это значение типа `short int`. Отображение слов на векторы позиций, построенное функцией `build_text_map()`, содержит такие векторы для каждого встречающегося в тексте слова, распознанного нашей системой. Ключами для этого отображения служат значения типа `string`, представляющие слова. Например, для текста

```
Alice Emma has long flowing red hair. Her Daddy says
when the wind blows through her hair, it looks almost alive,
like a fiery bird in flight. A beautiful fiery bird, he tells her,
magical but untamed. "Daddy, shush, there is no such thing,"
she tells him, at the same time wanting him to tell her more.
```

```
Shyly, she asks, "I mean, Daddy, is there?"
```

приведена часть отображения для некоторых слов, встречающихся неоднократно (слово – это ключ отображения; пары значений в скобках – элементы вектора позиций; отметим, что нумерация строк и колонок начинается с нуля):

```
bird ((2,3),(2,9))
daddy ((0,8),(3,3),(5,5))
fiery ((2,2),(2,8))
hair ((0,6),(1,6))
her ((0,7),(1,5),(2,12),(4,11))
him ((4,2),(4,8))
she ((4,0),(5,1))
tell ((2,11),(4,1),(4,10))
```

Однако такой вектор – это еще ответ на запрос. К примеру, слово `fiery` представлено двумя позициями, причем обе находятся в одной и той же строке.

Нам нужно вычислить множество неповторяющихся строк, соответствующих вектору позиций. Для этого можно, например, создать вектор, в который помещаются все номера строк, представленные в векторе позиций, а затем передать его обобщенному алгоритму `unique()`, который удалит все дубликаты (см. алгоритм `unique()` в Приложении). Оставшиеся строки должны быть расположены в порядке возрастания номеров. Чтобы не оставалось никаких сомнений, к вектору строк можно применить обобщенный алгоритм `sort()`.

Мы выбрали другой подход – построить множество (объект `set`) из номеров строк в векторе позиций. Такое множество содержит по одному экземпляру каждого элемента, причем хранит их в отсортированном виде. Нам потребуется функция для преобразования вектора позиций в множество неповторяющихся номеров строк:

```
set<short>* Query::_vec2set( const vector< location >* );
```

Объявим `_vec2set()` защищенной функцией-членом `Query`. Она не является открытой, поскольку не принадлежит к числу операций, которые могут вызывать пользователи данной иерархии. Но она и не закрыта, поскольку это вспомогательная функция, которая должна быть доступна производным классам. (Подчерк в имени функции призван обратить внимание на то, что это не часть открытого интерфейса иерархии `Query`.)

Например, вектор позиций для слова `bird` содержит два вхождения в одной и той же строке, поэтому его разрешающее множество будет состоять из одного элемента: (2). Вектор позиций для слова `tell` содержит три вхождения, из них два относятся к одной и той же строке; следовательно, в его разрешающем множестве будет два элемента: (2,4). Вот как выглядят результаты для всех представленных выше векторов позиций:

```
bird   (2)
daddy  (0,3,5)
fiery  (2)
hair   (0,1)
her    (0,1,2,4)
him    (4)
she    (4,5)
tell   (2,4)
```

Чтобы вычислить результат запроса NameQuery, достаточно получить вектор позиций для указанного слова, преобразовать его в множество неповторяющихся номеров строк и вывести соответствующие строки текста.

Ответом на NotQuery служит множество строк, в которых не встречается указанное слово. Так, результатом запроса

```
! daddy
```

служит множество (1,2,4). Для вычисления результата надо знать, сколько всего строк содержится в тексте. (Мы не сохраняли эту информацию, поскольку не были уверены, что она потребуется; к сожалению, недостаточно и этого.) Чтобы упростить обработку NotQuery, полезно сгенерировать множество всех номеров строк текста (0,1,2,3,4,5): теперь для получения результата достаточно с помощью алгоритма set_difference() вычислить разность двух множеств. (Ответом на показанный выше запрос будет множество (0,3,5).)

Результатом OrQuery является объединение номеров строк, где встречается левый или правый операнд. Например, если дан запрос:

```
fiery || her
```

то результирующим множеством будет (0,1,2,4), которое получается объединением множества (2) для слова fiery и множества (0,1,2,4) для слова her. Такое множество должно быть упорядочено по возрастанию номеров строк и не содержать дубликатов.

До сих пор нам удавалось вычислять результат запроса, работая только с множествами неповторяющихся номеров строк. Однако для обработки AndQuery надо принимать во внимание как номер строки, так и номер колонки в каждой паре. Так, указанные в запросе

```
her && hair
```

слова встречаются в четырех разных строках. Определенная нами семантика AndQuery говорит, что строка является подходящей, если содержит точную последовательность her hair. Вхождения слов в первую строку не удовлетворяют этому условию, хотя они стоят рядом:

```
Alice Emma has long flowing red hair. Her Daddy says
```

а вот во второй строке слова расположены так, как нужно:

```
when the wind blows through her hair, it looks almost alive,
```

Для оставшихся двух вхождений слова her слово hair не является соседним. Таким образом, ответом на запрос является вторая строка текста: (1).

Если бы не операция AndQuery, нам не пришлось бы вычислять вектор позиций для каждой операции. Но, поскольку операндом AndQuery может быть результат любого запроса, то для каждого приходится вычислять и сохранять не только множество неповторяющихся строк, но и пары (строка, колонка). Рассмотрим следующие запросы:

```
fiery && ( hair || bird || potato )
fiery && ( ! burr )
```

NotQuery может быть операндом AndQuery, следовательно, мы должны создать не просто вектор, содержащий по одному элементу для каждой подходящей строки, но и вектор, в котором хранятся позиции. (Мы еще вернемся к этому при рассмотрении функции eval() для класса NotQuery в разделе 17.5.)

Таким образом, идентифицирован еще один необходимый член – вектор позиций, ассоциированный с вычислением каждой операции. У нас есть выбор: объявить его членом каждого производного класса или членом абстрактного базового класса Query, наследуемым всеми производными. Объем памяти для хранения этого члена в обоих случаях одинаков. Мы поместим его в базовый класс, локализовав поддержку инициализации и доступа к члену.

Решение о том, представлять ли множество неповторяющихся номеров строк (мы называем его *разрешающим множеством*) в виде члена класса или каждый раз вычислять его, принимает разработчик. Мы предпочли вычислять его по мере необходимости, а затем сохранять адрес для последующего доступа, объявляя этот адрес членом абстрактного базового класса Query.

Для вывода найденных строк нам необходимо как разрешающее множество, так и фактический текст, из которого взяты строки. Причем вектор позиций у каждой операции должен быть свой, а экземпляр текста нужен только один. Поэтому мы определим его статическим членом класса Query. (Реализация функции display() опирается только на эти два члена.)

Вот результат первой попытки создать абстрактный базовый класс Query (конструкторы, деструктор и копирующий оператор присваивания еще не объявлены: этим мы займемся в разделах 17.4 и 17.6):

```
#include <vector>
#include <set>
#include <string>
#include <utility>

typedef pair< short, short > location;

class Query {
public:
     // конструкторы и деструктор обсуждаются в разделе 17.4

     // копирующий конструктор и копирующий оператор присваивания
     // обсуждаются в разделе 17.6

     // операции для поддержки открытого интерфейса
          virtual void eval() = 0;
          virtual void display () const;

     // функции доступа для чтения
          const set<short> *solution() const;
          const vector<location> *locations() const { return &_loc; }

     static const vector<string> *text_file() {return _text_file;}

protected:
          set<short>* _vec2set( const vector<location>* );

     static vector<string> *_text_file;

          set<short>         *_solution;
          vector<location>   _loc;
};

inline const set<short>
Query::
solution()
{
   return _solution
        ? _solution
        : _solution = _vec2set( &_loc );
}
```

Странный синтаксис

```
          virtual void eval() = 0;
```

говорит о том, что для виртуальной функции `eval()` в абстрактном базовом классе `Query` нет определения: это *чисто виртуальная функция*, "удерживающая место" в открытом интерфейсе иерархии классов и не предназначенная для непосредственного вызова из программы. Вместо нее каждый производный класс должен предоставить настоящую реализацию. (Подробно виртуальные функции будут рассматриваться в разделе 17.5.)

17.2.2. Определение производных классов

Каждый производный класс наследует данные и функции-члены своего базового класса, и программировать приходится лишь те аспекты, которые изменяют или расширяют его поведение. К примеру, в классе `NameQuery` необходимо определить реализацию `eval()`.

Кроме того, нужна поддержка для хранения слова-операнда, представленного объектом класса типа string.

Наконец, для получения ассоциированного вектора позиций должно быть доступно отображение слов на векторы. Поскольку один такой объект разделяется всеми объектами класса NameQuery, мы объявляем его статическим членом. Первая попытка определения NameQuery (рассмотрение конструкторов, деструктора и копирующего оператора

присваивания мы снова отложим) выглядит так:

```
typedef vector<location> loc;

class NameQuery : public Query {
public:
     // ...

     // переопределяет виртуальную функцию Query::eval()2
          virtual void eval();

     // функция чтения
          string name() const { return _name; }

     static const map<string,loc*> *word_map() { return _word_map; }

protected:
          string _name;
     static map<string,loc*> *_word_map;
};
```

Класс NotQuery в дополнение к предоставлению реализации виртуальной функции eval() должен обеспечить поддержку своего единственного операнда. Поскольку им может быть объект любого из производных классов, определим его как указатель на тип Query. Результат запроса NotQuery, напомним, обязан содержать не только строки текста, где нет указанного слова, но также и номера колонок внутри каждой строки. Например, если есть запрос:

```
! daddy
```

то операнд запроса NotQuery включает следующий вектор позиций:

```
daddy ((0,8),(3,3),(5,5))
```

Вектор позиций, возвращаемый в ответ на исходный запрос, должен включать все номера колонок в строках (1,2,4). Кроме того, он должен включать все номера колонок в строке (0), кроме колонки (8), все номера колонок в строке (3), кроме колонки (3), и все номера колонок в строке (5), кроме колонки (5).

---

2 В объявлении унаследованной виртуальной функции, например eval(), в производном классе ключевое слово virtual необязательно. Компилятор делает правильное заключение на основе сравнения с прототипом функции.

Простейший способ вычислить все это – создать единственный разделяемый всеми объектами вектор позиций, который содержит пары (строка, колонка) для каждого слова в тексте (полную реализацию мы рассмотрим в разделе 17.5, когда будем обсуждать функцию `eval()` класса `NotQuery`). Так или иначе, этот член мы объявим статическим для `NotQuery`.

Вот определение класса `NotQuery` (и снова рассмотрение конструкторов, деструктора и копирующего оператора присваивания отложено):

```
class NotQuery : public Query {
public:
     // ...

     // альтернативный синтаксис: явно употреблено ключевое слово virtual
     // переопределение Query::eval()
          virtual void eval();

     // функция доступа для чтения
          const Query *op() const { return _op; }
          static const vector< location > * all_locs() {
                                     return _all_locs; }

protected:
          Query *_op;
          static const vector< location > *_all_locs;
};
```

Классы `AndQuery` и `OrQuery` представляют бинарные операции, у которых есть левый и правый операнды. Оба операнда могут быть объектами любого из производных классов, поэтому мы определим соответствующие члены как указатели на тип `Query`. Кроме того, в каждом классе нужно переопределить виртуальную функцию `eval()`. Вот начальное определение `OrQuery`:

```
class OrQuery : public Query {
public:
     // ...

          virtual void eval();

          const Query *rop() const { return _rop; }
          const Query *lop() const { return _lop; }

protected:
          Query *_lop;
          Query *_rop;
};
```

Любой объект `AndQuery` должен иметь доступ к числу слов в каждой строке. В противном случае при обработке запроса `AndQuery` мы не сможем найти соседние слова, расположенные в двух смежных строках. Например, если есть запрос:

```
tell && her && magical
```

то нужная последовательность находится в третьей и четвертой строках:

```
like a fiery bird in flight. A beautiful fiery bird, he tells her,
magical but untamed. "Daddy, shush, there is no such thing,"
```

Векторы позиций, ассоциированные с каждым из трех слов, следующие:

```
her     ((0,7),(1,5),(2,12),(4,11))
magical ((3,0))
tell    ((2,11),(4,1),(4,10))
```

Если функция eval() класса AndQuery "не знает", сколько слов содержится в строке (2), то она не сможет определить, что слова magical и her соседствуют. Мы создадим единственный экземпляр вектора, разделяемый всеми объектами класса, и объявим его статическим членом. (Реализацию eval() мы детально рассмотрим в разделе 17.5.) Итак,

```
class AndQuery : public Query {
public:
     // конструкторы обсуждаются в разделе 17.4
          virtual void eval();

          const Query *rop() const { return _rop; }
          const Query *lop() const { return _lop; }

          static void max_col( const vector< int > *pcol )
                 { if ( !_max_col ) _max_col = pcol; }

protected:
          Query *_lop;
          Query *_rop;
          static const vector< int > *_max_col;
```

определим AndQuery:

```
};
```

17.2.3. Резюме

Открытый интерфейс каждого из четырех производных классов состоит из их открытых членов и унаследованных открытых членов Query. Когда мы пишем:

```
Query *pq = new NmaeQuery( "Monet" );
```

то получить доступ к открытому интерфейсу Query можно только через pq. А если пишем:

```
pq->eval();
```

то вызывается реализация виртуальной eval() из производного класса, на объект которого указывает pq, в данном случае – из класса NameQuery. Строкой

```
pq->display();
```

всегда вызывается невиртуальная функция display() из Query. Однако она выводит разрешающее множество строк объекта того производного класса, на который указывает pq. В этом случае мы не стали полагаться на механизм виртуализации, а вынесли разделяемую операцию и необходимые для нее данные в общий абстрактный базовый класс Query. display() – это пример полиморфного программирования, которое поддерживается не виртуальностью, а исключительно с помощью наследования. Вот ее реализация (это пока только промежуточное решение, как мы увидим в последнем

```
void
Query::
display()
{
   if ( ! _solution->size() ) {
      cout << "\n\tИзвините, "
           << " подходящих строк в тексте не найдено.\n"
           << endl;
   }

   set<short>::const_iterator
       it = _solution->begin(),
       end_it = _solution->end();

   for ( ; it != end_it; ++it ) {
      int line = *it;

      // не будем пользоваться нумерацией строк с 0...
      cout << "(" << line+1 << " ) "
           << (*_text_file)[line] << '\n';
   }

   cout << endl;
```

разделе):

```
}
```

В этом разделе мы попытались определить иерархию классов Query. Однако вопрос о том, как же построить с ее помощью структуру данных, описывающую запрос пользователя, остался без ответа. Когда мы приступим к реализации, это определение придется пересмотреть и расширить. Но прежде нам предстоит более детально изучить механизм наследования в языке C++.

Упражнение 17.3

Рассмотрите приведенные члены иерархии классов для поддержки библиотеки из упражнения 17.1 (раздел 17.1). Выявите возможные кандидаты на роль виртуальных функций, а также те члены, которые являются общими для всех предметов, выдаваемых библиотекой, и, следовательно, могут быть представлены в базовом классе. (Примечание: LibMember – это абстракция человека, которому разрешено брать из библиотеки различные предметы; Date – класс, представляющий календарную дату.)

```
class Library {
public:
   bool check_out( LibMember* );   // выдать
   bool check_in ( LibMember* );   // принять назад
   bool is_late( const Date& today );  // просрочил
   double apply_fine();                // наложить штраф
   ostream& print( ostream&=cout );

   Date* due_date() const;             // ожидаемая дата возврата
   Date* date_borrowed() const;        // дата выдачи

   string title() const;               // название
   const LibMember* member() const;    // записавшийся
};
```

Упражнение 17.4

Идентифицируйте члены базового и производных классов для той иерархии, которую вы выбрали в упражнении 17.2 (раздел 17.1). Задайте виртуальные функции, а также открытые и защищенные члены.

Упражнение 17.5

```
class base { ... };

(a) class Derived : public Derived { ... };
(b) class Derived : Base { ... };
(c) class Derived : private Base { ... };
(d) class Derived : public Base;
```

Какие из следующих объявлений неправильны:

```
(e) class Derived inherits Base { ... };
```

17.3. Доступ к членам базового класса

Объект производного класса фактически построен из нескольких частей. Каждый базовый класс вносит свою долю в виде подобъекта, составленного из нестатических данных-членов этого класса. Объект производного класса построен из подобъектов, соответствующих каждому из его базовых, а также из части, включающей нестатические члены самого производного класса. Так, наш объект `NameQuery` состоит из подобъекта `Query`, содержащего члены `_loc` и `_solution`, и части, принадлежащей `NameQuery`, – она содержит только член `_name`.

Внутри производного класса к членам, унаследованным из базового, можно обращаться напрямую, как к его собственным. (Глубина цепочки наследования не увеличивает затраты времени и не лимитирует доступ к ним.) Например:

```
void
NameQuery::
display_partial_solution( ostream &os )
{
   os << _name
      << " is found in "
      << (_solution ? _solution->size() : 0)
      << " lines of text\n";
}
```

Это касается и доступа к унаследованным функциям-членам базового класса: мы

```
NameQuery nq( "Frost" );

// вызывается NameQuery::eval()
nq.eval();

// вызывается Query::display()
```

вызываем их так, как если бы они были членами производного – либо через его объект:

```
nq.display();
```

```
void
NameQuery::
match_count()
{
   if ( ! _solution )
      // вызывается Query::_vec2set()
      _solution = _vec2set( &_loc );
   return _solution->size();
}
```

либо непосредственно из тела другой (или той же самой) функции-члена:

Однако прямой доступ из производного класса к членам базового запрещен, если имя

```
class Diffident {
public:  // ...
protected:
   int _mumble;
   // ...
};

class Shy : public Diffident {
public: // ...
protected:
   // имя Diffident::_mumble скрыто
   string _mumble;

   // ...
```

последнего скрыто в производном классе:

```
};
```

В области видимости Shy употребление неквалифицированного имени _mumble разрешается в пользу члена _mumble класса Shy (объекта string), даже если такое

```
void
Shy::
turn_eyes_down()
{
   // ...
   _mumble = "excuse me";   // правильно

   // ошибка: int Diffident::_mumble скрыто
   _mumble = -1;
}
```

использование в данном контексте недопустимо:

Некоторые компиляторы помечают это как ошибку типизации. Для доступа к члену базового класса, имя которого скрыто в производном, необходимо квалифицировать имя члена базового класса именем самого этого класса с помощью оператора разрешения области видимости. Так выглядит правильная реализация функции-члена

```
void
Shy::
turn_eyes_down()
{
   // ...
   _mumble = "excuse me";   // правильно

   // правильно: имя члена базового класса квалифицировано
   Diffident::_mumble = -1;
}
```

turn_eyes_down():

Функции-члены базового и производного классов не составляют множество

```
class Diffident {
public:
   void mumble( int softness );
   // ...
};

class Shy : public Diffident {
public:
   // скрывает видимость функции-члена Diffident::_mumble,
   // а не перегружает ее
   void mumble( string whatYaSay );
   void print( int soft, string words );
   // ...
};
```

перегруженных функций:

Вызов функции-члена базового класса из производного в этом случае приводит к ошибке компиляции:

```
Shy simon;

// правильно: Shy::mumble( string )
simon.mumble( "pardon me" );

// ошибка: ожидался первый аргумент типа string
// Diffident::mumble( int ) невидима

simon.mumble( 2 );
```

Хотя к членам базового класса можно обращаться напрямую, они сохраняют область видимости класса, в котором определены. А чтобы функции перегружали друг друга, они должны находиться в одной и той же области видимости. Если бы это было не так,

```
class Diffident {
public:
   void turn_aside( );
   // ...
};

class Shy : public Diffident {
public:
   // скрывает видимость
   // Diffident::turn_aside()
   void turn_aside();

   // ...
};
```

следующие два экземпляра невиртуальной функции-члена `turn_aside()`

привели бы к ошибке повторного определения, так как их сигнатуры одинаковы. Однако запись правильна, поскольку каждая функция находится в области видимости того класса, в котором определена.

А если нам действительно нужен набор перегруженных функций-членов базового и производного классов? Написать в производном классе небольшую встроенную заглушку

```
class Shy : public Diffident {
public:
   // один из способов реализовать множество перегруженных
   // членов базового и производного классов
   void mumble( string whatYaSay );
   void mumble( int softness ) {
                Diffident::mumble( softness ); }
   // ...
};
```

для вызова экземпляра из базового? Это возможно:

Но в стандартном C++ тот же результат достигается посредством using-объявления:

```
class Shy : public Diffident {
public:
   // в стандартном C++ using-объявление
   // создает множество перегруженных
   // членов базового и производного классов
   void mumble( string whatYaSay );
   using Diffident::mumble;

   // ...
};
```

По сути дела, using-объявление вводит каждый именованный член базового класса в область видимости производного. Поэтому такой член теперь входит в множество перегруженных функций, ассоциированных с именем функции-члена производного класса. (В ее using-объявлении нельзя указать список параметров, только имя. Это означает, что если некоторая функция уже перегружена в базовом классе, то в область видимости производного класса попадут все перегруженные экземпляры и, следовательно, добавить только одну из них невозможно.)

Обратим внимание на степень доступности защищенных членов базового класса. Когда

```
class Query {
public:
   const vector<location>* locations() { return &_loc; }
   // ...
protected:
   vector<location> _loc;
   // ...
};
```

мы пишем:

то имеем в виду, что класс, производный от Query, может напрямую обратиться к члену _loc, тогда как во всей остальной программе для этого необходимо пользоваться открытой функцией доступа. Однако объект производного класса имеет доступ только к защищенному члену _loc *входящего в него* подобъекта, относящегося к базовому классу. Объект производного класса неспособен обратиться к защищенным членам другого

```
bool
NameQuery::
compare( const Query *pquery )
{
   // правильно: защищенный член подобъекта Query
   int myMatches = _loc.size();

   // ошибка: нет прав доступа к защищенному члену
   // независимого объекта Query
   int itsMatches = pquery->_loc.size();

   return myMatches == itsMatches;
}
```

независимого объекта базового класса:

У объекта NameQuery есть доступ к защищенным членам только одного объекта Query – подобъекта самого себя. Прямое обращение к ним из производного класса осуществляется через неявный указатель this (см. раздел 13.4). Первая реакция на ошибку компиляции – переписать функцию compare() с использованием открытой функции-члена

```
bool
NameQuery::
compare( const Query *pquery )
{
   // правильно: защищенный член подобъекта Query
   int myMatches = _loc.size();

   // правильно: используется открытый метод доступа
   int itsMatches = pquery->locations()->size();

   return myMatches == itsMatches;
```

location():

```
}
```

Однако проблема заключается в неправильном проектировании. Поскольку _loc – это член базового класса Query, то место compare() среди членов базового, а не производного класса. Во многих случаях подобные проблемы могут быть решены путем переноса некоторой операции в тот класс, где находится недоступный член, как в приведенном примере.

Этот вид ограничения доступа не распространяется на доступ изнутри класса к другим

```
bool
NameQuery::
compare( const NameQuery *pname )
{
   int myMatches = _loc.size();   // правильно
   int itsMatches = name->_loc.size();   // тоже правильно

   return myMatches == itsMatches;
```

объектам того же класса:

```
}
```

Производный класс может напрямую обращаться к защищенным членам базового в других объектах того же класса, что и он сам, равно как и к защищенным и закрытым членам других объектов своего класса.

Рассмотрим инициализацию указателя на базовый Query адресом объекта производного NameQuery:

```
Query *pb = new NameQuery( "sprite" );
```

При вызове виртуальной функции, определенной в базовом классе Query, например:

```
pb->eval();  // вызывается NameQuery::eval()
```

вызывается функция из NameQuery. За исключением вызова виртуальной функции, объявленной в Query и переопределенной в NameQuery, другого способа напрямую добраться до членов класса NameQuery через указатель pb не существует:

(a) если в Query и NameQuery объявлены некоторые невиртуальные функции-члены с одинаковым именем, то через pb всегда вызывается экземпляр из Query;

(b) если в Query и NameQuery объявлены одноименные члены, то через pb обращение происходит к члену класса Query;

(c) если в NameQuery имеется виртуальная функция, отсутствующая в Query, скажем suffix(), то попытка вызвать ее через pb приводит к ошибке

```
// ошибка: suffix() - не член класса Query
```

компиляции:

```
pb->suffix();
```

- Обращение к члену или невиртуальной функции-члену класса NameQuery через

```
// ошибка: _name - не член класса Query
```

pb тоже вызывает ошибку компиляции:

```
pb->_name;
```

```
// ошибка: у класса Query нет базового класса NameQuery
```

Квалификация имени члена в этом случае не помогает:

```
pb->NameQuery::_name;
```

В C++ с помощью указателя на базовый класс можно работать только с данными и функциями-членами, включая виртуальные, которые объявлены (или унаследованы) в самом этом классе, независимо от того, какой фактический объект адресуется указателем. Объявление функции-члена виртуальной откладывает решение вопроса о том, какой экземпляр функции вызвать, до выяснения (во время выполнения программы) фактического типа объекта, адресуемого pb.

Такой подход может показаться недостаточно гибким, но у него есть два весомых преимущества:

- поиск виртуальной функции-члена во время выполнения никогда не закончится неудачно из-за того, что фактический тип класса не существует. В таком случае программа просто не смогла бы откомпилироваться;
- механизм виртуализации можно оптимизировать. Часто вызов такой функции оказывается не дороже, чем косвенный вызов функции по указателю (детально этот вопрос рассмотрен в [LIPPMAN96a]).

В базовом классе Query определен статический член _text_file:

```
static vector<string> *_text_file;
```

Создается ли при порождении класса NameQuery второй экземпляр _text_file, уникальный именно для него? Нет. Все объекты производного класса ссылаются на тот же самый, единственный разделяемый статический член. Сколько бы ни было производных классов, существует лишь один экземпляр _text_file. Можно обратиться к нему через объект производного класса с помощью синтаксиса доступа:

```
nameQueryObject._text_file;   // правильно
```

Наконец, если производный класс хочет получить доступ к закрытым членам своего

```
class Query {
   friend class NameQuery;
public:
   // ...
```

базового класса напрямую, то он должен быть объявлен другом базового:

```
};
```

Теперь объект NameQuery может обращаться не только к закрытым членам своего подобъекта, соответствующего базовому классу, но и к закрытым и защищенным членам любых объектов Query.

А если мы произведем от NameQuery класс StringQuery? Он будет поддерживать сокращенную форму запроса AndQuery, и вместо

```
beautiful && fiery && bird
```

можно будет написать:

```
"beautiful fiery bird"
```

Унаследует ли StringQuery от класса NameQuery дружественные отношения с Query? Нет. Отношение дружественности не наследуется. Производный класс не становится другом класса, который объявил своим другом один из базовых. Если производному классу требуется стать другом одного или более классов, то эти классы должны предоставить ему соответствующие права явно. Например, у класса StringQuery нет никаких специальных прав доступа по отношению к Query. Если расширенный доступ необходим, то Query должен разрешить его явно.

Упражнение 17.6

Даны следующие определения базового и производных классов:

```
class Base {
public:
   foo( int );
   // ...
protected:
   int _bar;
   double _foo_bar;
};

class Derived : public Base {
public:
   foo( string );
   bool bar( Base *pb );
   void foobar();
   // ...
protected:
   string _bar;
};
```

```
Derived d; d.foo( 1024 );
```

Исправьте ошибки в каждом из следующих фрагментов кода:

```
(c) bool Derived::bar( Base *pb )
```

```
 (b) void Derived::foobar() { _bar = 1024; }

         { return _foo_bar == pb->_foo_bar; }
```

17.4. Конструирование базового и производного классов

Напомним, что объект производного класса состоит из одного или более подобъектов, соответствующих базовым классам, и части, относящейся к самому производному. Например, `NameQuery` состоит из подобъекта `Query` и объекта-члена `string`. Для иллюстрации поведения конструктора производного класса введем еще один член

```
class NameQuery : public Query {
public:
   // ...
protected:
   bool   _present;
   string _name;
};
```

встроенного типа:

Если `_present` установлен в `false`, то слово `_name` в тексте отсутствует.

Рассмотрим случай, когда в NameQuery конструктор не определен. Тогда при определении объекта этого класса

```
NameQuery nq;
```

по очереди вызывается конструктор по умолчанию Query, а затем конструктор по умолчанию класса string (ассоциированный с объектом _name). Член _present остается неинициализированным, что потенциально может служить источником ошибок. Чтобы инициализировать его, можно так определить конструктор по умолчанию для класса NameQuery:

```
inline NameQuery::NameQuery() { _present = false; }
```

Теперь при определении nq вызываются три конструктора по умолчанию: для базового класса Query, для класса string при инициализации члена _name и для класса NameQuery.

А как передать аргумент конструктору базового класса Query? Ответить на этот вопрос можно, рассуждая по аналогии.

Для передачи одного или более аргументов конструктору объекта-члена мы используем список инициализации членов (здесь можно также задать начальные значения членам, не

```
inline NameQuery::
NameQuery( const string &name )
         : _name( name ), _present( false )
```

являющимся объектами классов; подробности см. в разделе 14.5):

```
{}
```

Для передачи одного или более аргументов конструктору базового класса также разрешается использовать список инициализации членов. В следующем примере мы передаем конструктору string аргумент name, а конструктору базового класса Query –

```
inline NameQuery::
NameQuery( const string &name,
           vector<location> *ploc )
         : _name( name ), Query( *ploc ), _present( true )
```

объект, адресованный указателем ploc:

```
{}
```

Хотя Query помещен в список инициализации вторым, его конструктор всегда вызывается раньше конструктора для _name. Порядок их вызова следующий:

Конструктор базового класса. Если базовых классов несколько, то конструкторы вызываются в порядке их следования в списке базовых классов, а не в порядке появления в списке инициализации. (О множественном наследовании в этой связи мы поговорим в главе 18.)

Конструктор объекта-члена. Если в классе есть несколько таких членов, то конструкторы вызываются в порядке их объявления в классе, а не в порядке появления в списке инициализации (подробнее см. раздел 14.5).

Конструктор производного класса.

Конструктор производного класса должен стремиться передать значение члена базового класса подходящему конструктору того же класса, а не присваивать его напрямую. В противном случае реализации двух классов становятся *сильно связанными* и тогда изменить или расширить реализацию базового будет затруднительно. (Ответственность разработчика базового класса ограничивается предоставлением подходящего множества конструкторов.)

В оставшейся части этого раздела мы последовательно изучим конструктор базового класса и конструкторы четырех производных от него, а после этого рассмотрим альтернативный дизайн иерархии классов `Query`, чтобы познакомиться с иерархиями глубиной больше двух. В конце раздела речь пойдет о деструкторах классов.

17.4.1. Конструктор базового класса

```
class Query {
public:
   // ...
protected:
   set<short> *_solution;
   vector<location> _loc;
   // ...
```

В нашем базовом классе объявлено два нестатических члена: `_solution` и `_loc`:

```
};
```

Конструктор `Query` по умолчанию должен явно инициализировать только член `_solution`. Для инициализации `_loc` автоматически вызывается конструктор класса `vector`. Вот реализация нашего конструктора:

```
inline Query::Query(): _solution( 0 ) {}
```

В `Query` нам понадобится еще один конструктор, принимающий ссылку на вектор

```
inline
Query::
Query( const vector< locaton > &loc )
     : _solution( 0 ), _loc( loc )
```

позиций:

```
{}
```

Он вызывается только из конструктора `NameQuery`, когда объект этого класса используется для представления указанного в запросе слова. В таком случае передается предварительно подготовленный для него вектор позиций. Остальные три производных класса вычисляют свои векторы позиций в соответствующей функции-члене `eval()`. (В следующем подразделе мы покажем, как это делается. Реализации функций-членов `eval()` приведены в разделе 17.5.)

Какой уровень доступа обеспечить для конструкторов? Мы не хотим объявлять их открытыми, так как предполагается, что `Query` будет существовать в программе только в

виде подобъекта в составе объектов производных от него классов. Поэтому мы объявим

```
class Query {
public:
   // ...
protected:
   Query();
   // ...
```

конструктор не открытым, а защищенным:

```
};
```

Ко второму конструктору класса `Query` предъявляются еще более жесткие требования: он не только должен конструировать `Query` в виде подобъекта производного класса, но этот производный класс должен к тому же быть `NameQuery`. Можно объявить конструктор закрытым, а `NameQuery` сделать другом класса `Query`. (В предыдущем разделе мы говорили, что производный класс может получить доступ только к открытым и защищенным членам базового. Поэтому любая попытка вызвать второй конструктор из

```
class Query {
public:
   // ...
protected:
   Query();
   // ...
private:
   explicit Query( const vector<location>& );
```

классов `AndQuery`, `OrQuery` или `NotQuery` приведет к ошибке компиляции.)

```
};
```

(Необходимость второго конструктора спорна; вероятно, правильнее заполнить `_loc` в функции `eval()` класса `NameQuery`. Однако принятый подход в большей степени отвечает нашей цели проиллюстрировать использование конструктора базового класса.)

17.4.2. Конструктор производного класса

В классе `NameQuery` также определены два конструктора. Они объявлены открытыми,

```
class NameQuery : public Query {
public:
   explicit NameQuery( const string& );
   NameQuery( const string&, const vector<location>* );
   // ...
protected:
   // ...
```

поскольку ожидается, что в приложении будут создаваться объекты этого класса:

```
};
```

Конструктор с одним параметром принимает в качестве аргумента строку. Она передается конструктору объекта типа string, который вызывается для инициализации

```
inline
NameQuery::
NameQuery( const string &name )
   // Query::Query() вызывается неявно
   : _name( name )
```

члена _name. Конструктор по умолчанию базового класса Query вызывается неявно:

```
{}
```

Конструктор с двумя параметрами также принимает строку в качестве одного из них. Второй его параметр – это указатель на вектор позиций. Он передается закрытому конструктору базового класса Query. (Обратите внимание, что _present нам больше не

```
inline
NameQuery::
NameQuery( const string &name, vector<location> *ploc )
         : _name( name ), Query( *ploc )
```

нужен, и мы исключили его из числа членов NameQuery.)

```
{}
```

```
string title( "Alice" );
NameQuery *pname;

// проверим, встречается ли "Alice" в отображении слов
// если да, получить ассоциированный с ним вектор позиций

if ( vector<location> *ploc = retrieve_location( title ))
     pname = new NameQuery( title, ploc );
```

Конструкторы можно использовать так:

```
else pname = new NameQuery( title );
```

В каждом из классов NotQuery, OrQuery и AndQuery определено по одному

```
inline NotQuery::
NotQuery( Query *op = 0 ) : _op( op ) {}

inline OrQuery::
OrQuery( Query *lop = 0, Query *rop = 0 )
       : _lop( lop ), _rop( rop )
{}

inline AndQuery::
AndQuery( Query *lop = 0, Query *rop = 0 )
       : _lop( lop ), _rop( rop )
```

конструктору, каждый из которых вызывает конструктор базового класса неявно:

```
{}
```

(В разделе 17.7 мы построим объекты каждого из производных классов для представления различных запросов пользователя.)

17.4.3. Альтернативная иерархия классов

Хотя наша иерархия классов `Query` представляется вполне приемлемой, она вовсе не является единственно возможной. Например, `AndQuery` и `OrQuery` связаны с бинарной операцией, поэтому они в какой-то степени дублируют друг друга. Можно вынести все данные и функции-члены, общие для них, в абстрактный базовый класс `BinaryQuery`. Поддерево новой иерархии `Query` изображено на рисунке 17.2:

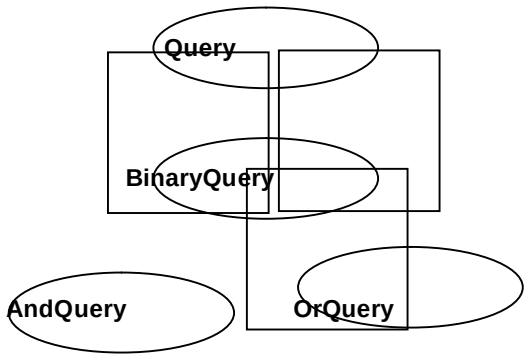


Рис. 17.2. Альтернативная иерархия классов

Класс `BinaryQuery` – это тоже абстрактный базовый класс, следовательно, его фактические объекты в приложении не появляются. Разумной реализации `eval()` для него предложить нельзя, поэтому чисто виртуальная функция, объявленная в `Query`, в классе `BinaryQuery` останется чисто виртуальной. (Подробнее о таких функциях мы поговорим в разделе 17.5.)

Две функции-члена для доступа – `lop()` и `rop()`, общие для обоих классов, переносятся выше, в `BinaryQuery`, и определяются как нестатические встроенные. Аналогично два члена `_lop` и `_rop`, объявленные в обоих классах, также переносятся в `BinaryQuery` и становятся нестатическими и защищенными. Открытые конструкторы обоих

```
class BinaryQuery : public Query {
public:
   const Query *lop() { return _lop; }
   const Query *rop() { return _rop; }

protected:
   BinaryQuery( Query *lop, Query *rop )
              : _lop( lop ), _rop( rop )
   {}

   Query *_lop;
   Query *_rop;
};
```

производных классов объединяются в один защищенный конструктор `BinaryQuery`:

Складывается впечатление, что теперь оба производных класса должны предоставить

```
// увы! эти определения классов некорректны

class OrQuery : public BinaryQuery {
public:
   virtual void eval();
};

class AndQuery : public BinaryQuery {
public:
   virtual void eval();
};
```

лишь подходящие реализации `eval()`:

```
};
```

Однако в том виде, в котором мы их определили, эти классы неполны. При компиляции самих определений ошибок не возникает, но если мы попытаемся определить

```
// ошибка: отсутствует конструктор класса AndQuery
AndQuery proust( new NameQuery( "marcel" ),
```

фактический объект:

```
                  new NameQuery( "proust " ));
```

то компилятор выдаст сообщение об ошибке: в классе `AndQuery` нет конструктора, готового принять два аргумента.

Мы предположили, что `AndQuery` и `OrQuery` наследуют конструктор `BinaryQuery` точно так же, как они наследуют функции-члены `lop()` и `rop()`. Однако производный класс не наследует конструкторов базового. (Это могло бы привести к ошибкам, связанным с неинициализированными членами производного класса. Представьте, что будет, если в `AndQuery` добавить пару членов, не являющихся объектами классов: унаследованный конструктор базового класса для инициализации объекта производного `AndQuery` применять уже нельзя. Однако программист может этого не осознавать. Ошибка проявится не при конструировании объекта `AndQuery`, а позже, при его использовании. Кстати говоря, перегруженные операторы `new` и `delete` наследуются, что иногда приводит к аналогичным проблемам.)

Каждый производный класс должен предоставлять собственный набор конструкторов. В случае классов `AndQuery` и `OrQuery` единственная цель конструкторов – обеспечить интерфейс для передачи двух своих операндов конструктору `BinaryQuery`. Так выглядит исправленная реализация:

```
// правильно: эти определения классов корректны

class OrQuery : public BinaryQuery {
public:
   OrQuery( Query *lop, Query *rop )
          : BinaryQuery( lop, rop ) {}

   virtual void eval();
};

class AndQuery : public BinaryQuery {
public:
   AndQuery( Query *lop, Query *rop )
          : BinaryQuery( lop, rop ) {}

   virtual void eval();

};
```

Если мы еще раз взглянем на рис. 17.2, то увидим, что `BinaryQuery` – непосредственный базовый класс для `AndQuery` и `OrQuery`, а `Query` –для `BinaryQuery`. Таким образом, `Query` не является непосредственным базовым классом для `AndQuery` и `OrQuery`.

Конструктору производного класса разрешается напрямую вызывать только конструктор своего непосредственного предшественника в иерархии (виртуальное наследование является исключением из этого правила, да и из многих других тоже: см. раздел 18.5). Например, попытка включить конструктор `Query` в список инициализации членов объекта `AndQuery` приведет к ошибке.

При определении объектов классов `AndQuery` и `OrQuery` теперь вызываются три конструктора: для базового `Query`, для непосредственного базового класса `BinaryQuery` и для производного `AndQuery` или `OrQuery`. (Порядок вызова конструкторов базовых классов отражает обход дерева иерархии наследования в глубину.) Дополнительный уровень иерархии, связанный с `BinaryQuery`, практически не влияет на производительность, поскольку мы определили его конструкторы как встроенные.

Так как модифицированная иерархия сохраняет открытый интерфейс исходного проекта, то все эти изменения не сказываются на коде, который был написан в расчете на старую иерархию. Хотя модифицировать пользовательский код не нужно, перекомпилировать его все же придется, что может отвратить некоторых пользователей от перехода на новую версию.

17.4.4. Отложенное обнаружение ошибок

Начинающие программисты часто удивляются, почему некорректные определения классов `AndQuery` и `OrQuery` (в которых отсутствуют необходимые объявления конструкторов) компилируются без ошибок. Если бы мы не попытались определить фактический объект класса `AndQuery`, в этой модифицированной иерархии так и осталась бы ненайденная ошибка. Дело в том, что:

- если ошибка обнаруживается в точке объявления, то мы не можем продолжать компиляцию приложения, пока не исправим ее. Если же конфликтующее объявление – это часть библиотеки, для которой у нас нет исходного текста, то разрешение конфликта может оказаться нетривиальной задачей. Более того, возможно, в нашем коде никогда и не возникнет ситуации, когда эта ошибка проявляется, так что для нас она останется лишь потенциальной угрозой;

- с другой стороны, если ошибка не найдена вплоть до момента использования, то код может оказаться замусоренным ошибками, проявляющимися в самый неподходящий момент к удивлению программиста. При такой стратегии успешная компиляция говорит не об отсутствии семантических ошибок, а лишь о том, что программа не исполняет код, нарушающий семантические правила языка.

Выдача сообщения об ошибке в точке использования – это одна из форм *отложенного вычисления*, распространенного метода повышения производительности программ. Он часто применяется для того, чтобы отложить потенциально дорогую операцию выделения или инициализации ресурса до момента, когда в нем возникнет реальная необходимость. Если ресурс так и не понадобится, мы сэкономим на ненужных подготовительных операциях. Если же он потребуется, но не сразу, мы растянем инициализацию программы на более длительный период.

В C++ потенциальные ошибки "комбинирования", связанные с перегруженными функциями, шаблонами и наследованием классов, обнаруживаются в точке использования, а не в точке объявления. (Мы полагаем, что это правильно, поскольку необходимость выявлять все возможные ошибки, которые можно допустить в результате комбинирования многочисленных компонентов, – пустая трата времени). Следовательно, для обнаружения и устранения латентных ошибок необходимо тщательно тестировать код. Подобные ошибки, возникающие при комбинировании двух или более больших компонентов, допустимы; однако в пределах одного компонента, такого, как иерархия классов `Query`, их быть не должно.

17.4.5. Деструкторы

Когда заканчивается время жизни объекта производного класса, автоматически вызываются деструкторы производного и базового классов (если они определены), а также деструкторы всех объектов-членов. Например, если имеется объект класса `NameQuery`:

```
NameQuery nq( "hyperion" );
```

то порядок вызова деструкторов следующий: сначала деструктор `NameQuery`, затем деструктор `string` для члена `_name` и наконец деструктор базового класса. В общем случае эта последовательность противоположна порядку вызова конструкторов.

Вот деструкторы нашего базового `Query` и производных от него (все они объявлены

```
inline Query::
~Query(){ delete _solution; }

inline NotQuery::
~NotQuery(){ delete _op; }

inline OrQuery::
~OrQuery(){ delete _lop; delete _rop; }

inline AndQuery::
```

открытыми членами соответствующих классов):

```
~AndQuery(){ delete _lop; delete _rop; }
```

Отметим два аспекта:

- мы не предоставляем явного деструктора NameQuery, потому что никаких специальных действий по очистке его объекта предпринимать не нужно. Деструкторы базового класса и класса string для члена _name вызываются автоматически;
- в деструкторах производных классов оператор delete применяется к указателю типа Query*. Чтобы вызвать не деструктор Query, а деструктор класса того объекта, который фактически адресуется этим указателем, мы должны объявить деструктор базового Query виртуальным. (Более подробно о виртуальных функциях вообще и о виртуальных деструкторах в частности мы поговорим в следующем разделе.)

В нашей реализации неявно подразумевалось, что память для операндов, указатели на которые имеются в объектах классов NotQuery, OrQuery и AndQuery, выделена из хипа. Именно поэтому в деструкторах мы применяли к этим указателям оператор delete. Но язык не позволяет обеспечить истинность такого предположения, так как в нем нет различий между адресами в хипе и вне его. С этой точки зрения наша реализация не застрахована от ошибок.

В разделе 17.7 мы инкапсулируем выделение памяти и конструирование объектов иерархии Query в управляющий класс UserQuery. Это гарантирует выполнение нашего предположения. На уровне программы в целом следует перегрузить операторы new и delete для классов иерархии. Например, можно поступить следующим образом. Оператор new устанавливает в объекте флажок, говорящий, что память для него выделена из хипа. Перегруженный оператор delete проверяет этот флажок: если он есть, то память освобождается с помощью стандартного оператора delete.

Упражнение 17.7

Идентифицируйте конструкторы и деструкторы базового и производных классов для той иерархии, которую вы выбрали в упражнении 17.2 (раздел 17.1).

Упражнение 17.8

Измените реализацию класса OrQuery так, чтобы он был производным от BinaryQuery.

Упражнение 17.9

```
class Object {
public:
   virtual ~Object();
   virtual string isA();
protected:
   string _isA;
private:
   Object( string s ) : _isA( s ) {}
```

Найдите ошибку в следующем определении класса:

```
};
```

Упражнение 17.10

Дано определение базового класса:

```
class ConcreteBase {
public:
   explicit ConcreteBase( int );
   virtual ostream& print( ostream& );
   virtual ~Base();

   static int object_count();
protected:
   int _id;
   static int _object_count;

};
```

```
(a) class C1 : public ConcreteBase {
    public:
       C1( int val )
         : _id( _object_count++ ) {}
       // ...
```

Что неправильно в следующих фрагментах:

```
(b) class C2 : public C1 {
    public:
       C2( int val )
         : ConcreteBase( val ), C1( val ) {}
       // ...

    };
```

```
(c) class C3 : public C2 {
    public:
       C3( int val )
         : C2( val ), _object_count( val ) {}
       // ...

    };
```

```
(d) class C4 : public ConcreteBase {
    public:
       C4( int val )
         : ConcreteBase ( _id+val ){}
       // ...

    };

    };
```

Упражнение 17.11

В первоначальном определении языка C++ порядок следования инициализаторов в списке инициализации членов определял порядок вызова конструкторов. Принцип,

который действует сейчас, был принят в 1986 году. Как вы думаете, почему была изменена исходная спецификация?

17.5. Виртуальные функции в базовом и производном классах

По умолчанию функции-члены класса не являются виртуальными. В подобных случаях при обращении вызывается функция, определенная в статическом типе объекта класса

```
void Query::display( Query *pb )
{
   set<short> *ps = pb->solutions();
   // ...
   display();
```

(или указателя, или ссылки на объект), для которого она вызвана:

```
}
```

Статический тип pb – это Query*. При обращении к невиртуальному члену solutions() вызывается функция-член класса Query. Невиртуальная функция display() вызывается через неявный указатель this. Статическим типом указателя this также является Query*, поэтому вызвана будет функция-член класса Query.

```
class Query {
public:
   virtual ostream& print( ostream* = cout ) const;
   // ...
```

Чтобы объявить функцию виртуальной, нужно добавить ключевое слово virtual:

```
};
```

Если функция-член виртуальна, то при обращении к ней вызывается функция, определенная в динамическом типе объекта класса (или указателя, или ссылки на объект), для которого она вызвана. Однако для самих объектов класса статический и динамический тип – это одно и то же. Механизм виртуальных функций правильно работает только для указателей и ссылок на объекты.

Таким образом, полиморфизм проявляется только тогда, когда объект производного класса адресуется косвенно, через указатель или ссылку на базовый. Использование самого объекта базового класса не сохраняет идентификацию типа производного.

```
NameQuery nq( "lilacs" );

// правильно: но nq "усечено" до подобъекта Query
```

Рассмотрим следующий фрагмент кода:

```
Query qobject = nq;
```

Инициализация qobject переменной nq абсолютно законна: теперь qobject равняется подобъекту nq, который соответствует базовому классу Query, однако qobject не

является объектом NameQuery. Часть nq, принадлежащая NameQuery, “усечена” перед инициализацией qobject, поскольку она не помещается в область памяти, отведенную под объект Query. Для поддержки этой парадигмы приходится использовать указатели и

```
void print ( Query object,
             const Query *pointer,
             const Query &reference )
{
   // до момента выполнения невозможно определить,
   // какой экземпляр print() вызывается
   pointer->print();
   reference.print();

   // всегда вызывается Query::print()
   object.print();
}

int main()
{
   NameQuery firebird( "firebird" );
   print( firebird, &firebird, firebird );
}
```

ссылки, но не сами объекты:

В данном примере оба обращения через указатель pointer и ссылку reference разрешаются своим динамическим типом; в обоих случаях вызывается NameQuery::print(). Обращение же через объект object всегда приводит к вызову Query::print(). (Пример программы, в которой используется эффект “усечения”, приведен в разделе 18.6.2.)

В следующих подразделах мы продемонстрируем определение и использование виртуальных функций в разных обстоятельствах. Каждая такая функция-член будет иллюстрировать один из аспектов объектно-ориентированного проектирования.

17.5.1. Виртуальный ввод/вывод

Первая виртуальная операция, которую мы хотели реализовать, – это печать запроса на стандартный вывод либо в файл:

```
ostream& print( ostream &os = cout ) const;
```

Функцию print() следует объявить виртуальной, поскольку ее реализации зависят от типа, но нам нужно вызывать ее через указатель типа Query*. Например, для класса

```
ostream&
AndQuery::print( ostream &os ) const
{
   _lop->print( os );
   os << " && ";
   _rop->print( os );
```

AndQuery эта функция могла бы выглядеть так:

```
}
```

Необходимо объявить print() виртуальной функцией в абстрактном базовом Query, иначе мы не сможем вызвать ее для членов классов AndQury, OrQuery и NotQuery, являющихся указателями на операнды соответствующих запросов типа Query*. Однако для самого Query разумной реализации print() не существует. Поэтому мы определим

```
class Query {
public:
   virtual ostream& print( ostream &os=cout ) const {}
   // ...
};
```

ее как пустую функцию, а потом сделаем чисто виртуальной:

В базовом классе, где виртуальная функция появляется в первый раз, ее объявлению должно предшествовать ключевое слово virtual. Если же ее определение находится вне этого класса, повторно употреблять virtual не следует. Так, данное определение

```
// ошибка: ключевое слово virtual может появляться
//         только в определении класса
virtual ostream& Query::print( ostream& ) const { ... }
```

print() приведет к ошибке компиляции:

Правильный вариант не должен включать слово virtual.

Класс, в котором впервые появляется виртуальная функция, должен определить ее или объявить чисто виртуальной (напомним, что пока мы определили ее как пустую). В производном классе может быть либо определена собственная реализация той же функции, которая в таком случае становится активной для всех объектов этого класса, либо унаследована реализация из базового класса. Если в производном классе определена собственная реализация, то говорят, что она *замещает* реализацию из базового.

Прежде чем приступать к рассмотрению реализаций print() для наших четырех производных классов, обратим внимание на употребление скобок в запросе. Например, с помощью

```
fiery && bird || shyly
```

пользователь ищет вхождения пары слов

```
fiery bird
```

или одного слова

```
shyly
```

С другой стороны, запрос

```
fiery && ( bird || hair )
```

найдет все вхождения любой из пар

```
fiery  bird
```

или

```
fiery hair
```

Если наши реализации `print()` не будут показывать скобки в исходном запросе, то для пользователя они окажутся почти бесполезными. Чтобы сохранить эту информацию, введем в наш абстрактный базовый класс `Query` два нестатических члена, а также функции доступа к ним (подобное расширение класса – естественная часть эволюции

```
class Query {
public:
     // ...

     // установить _lparen и _rparen
          void lparen( short lp ) { _lparen = lp; }
          void rparen( short rp ) { _rparen = rp; }

     // получить значения_lparen и _rparen
          short lparen() { return _lparen; }
          short rparen() { return _rparen; }

     // напечатать левую и правую скобки
          void print_lparen( short cnt, ostream& os ) const;
          void print_rparen( short cnt, ostream& os ) const;

protected:

     // счетчики левых и правых скобок
          short _lparen;
          short _rparen;
     // ...
```

иерархии):

```
};
```

`_lparen` – это количество левых, а `_rparen` – правых скобок, которое должно быть выведено при распечатке объекта. (В разделе 17.7 мы покажем, как вычисляются такие величины и как происходит присваивание обоим членам.) Вот пример обработки запроса с большим числом скобок:

```
==> ( untamed || ( fiery || ( shyly ) ) )
evaluate word: untamed
_lparen: 1
_rparen: 0
evaluate Or
_lparen: 0
_rparen: 0
```

```
evaluate word: fiery
_lparen: 1
_rparen: 0

evaluate 0r
_lparen: 0
_rparen: 0

evaluate word: shyly
_lparen: 1
_rparen: 0

evaluate right parens:
_rparen: 3

( untamed ( 1 ) lines match
( fiery ( 1 ) lines match
( shyly ( 1 ) lines match
( fiery || (shyly ( 2 ) lines match3
( untamed || ( fiery || ( shyly ))) ( 3 ) lines match

Requested query: ( untamed || ( fiery || ( shyly ) ) )
( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
( 4 ) magical but untamed. "Daddy, shush, there is no such thing,"
( 6 ) Shyly, she asks, "I mean, Daddy, is there?"
```

```
ostream&
NameQuery::
print( ostream &os ) const
{
   if ( _lparen )
      print_lparen( _lparen, os );

   os << _name;

   if ( _rparen )
      print_rparen( _rparen, os );

   return os;
```

Реализация `print()` для класса `NameQuery`:

```
}
```

```
class NameQuery : public Query {
public:
   virtual ostream& print( ostream &os ) const;
   // ...
```

А так выглядит объявление:

```
};
```

Чтобы реализация виртуальной функции в производном классе замещала реализацию из базового, прототипы функций обязаны совпадать. Например, если бы мы опустили слово `const` или объявили еще один параметр, то реализация `print()` в `NameQuery` не заместила бы реализацию из базового класса. Возвращаемые значения также должны

---

3 Увы! Правые скобки не распознаются, пока OrQuery не выведет все ассоциированное с ним частичное решение.

быть одинаковыми за одним исключением: значение, возвращенное реализацией в производном классе, может принадлежать к типу класса, который открыто наследует классу значения, возвращаемого реализацией в базовом классе. Если бы реализация из базового класса возвращала значение типа `Query*`, то реализация из производного могла бы возвращать `NameQuery*`. (Позже при работе с функцией `clone()` мы покажем, зачем

```
class NotQuery : public Query {
public:
   virtual ostream& print( ostream &os ) const;
   // ...
```

это нужно.) Вот объявление и реализация `print()` в `NotQuery`:

```
ostream&
NotQuery::
print( ostream &os ) const
{
   os << " ! ";

   if ( _lparen )
      print_lparen( _lparen, os );

   _op->print( os );

   if ( _rparen )
      print_rparen( _rparen, os );

   return os;

};
}
```

Разумеется, вызов `print()` через `_op` – виртуальный.

Объявления и реализации этой функции в классах `AndQuery` и `OrQuery` практически

```
class AndQuery : public Query {
public:
   virtual ostream& print( ostream &os ) const;
   // ...
```

дублируют друг друга. Поэтому приведем их только для `AndQuery`:

```
};
```

```
ostream&
AndQuery::
print( ostream &os ) const
{
   if ( _lparen )
           print_lparen( _lparen, os );

   _lop->print( os );
   os << " && ";
   _rop->print( os );

   if ( _rparen )
           print_rparen( _rparen, os );

   return os;
}
```

Такая реализация виртуальной функции `print()` позволяет вывести любой подтип `Query`

```
cout << "Был сформулирован запрос ";
Query *pq = retrieveQuery();
```

в поток класса `ostream` или любого другого, производного от него:

```
pq->print( cout );
```

Однако такой возможности недостаточно. Еще нужно уметь распечатывать любой производный от `Query` тип, который уже есть или может появиться в будущем, с

```
Query *pq = retrieveQuery();
cout << "В ответ на запрос "
     << *pq
```

помощью оператора вывода из библиотеки `iostream`:

```
     << " получены следующие результаты:\n";
```

Мы не можем непосредственно предоставить виртуальный оператор вывода, поскольку они являются членами класса `ostream`. Вместо этого мы должны написать косвенную

```
inline ostream&
operator<<( ostream &os, const Query &q )
{
   // виртуальный вызов print()
   return q.print( os );
```

виртуальную функцию:

```
}
```

```
AndQuery query;
// сформулировать запрос ...
```

Строки

```
cout << query << endl;
```

вызывают наш оператор вывода в ostream, который в свою очередь вызывает

```
q.print( os )
```

где q привязано к объекту query класса AndQuery, а os – к cout. Если бы вместо этого

```
NameQuery query2( "Salinger" );
```

мы написали:

```
cout << query2 << endl;
```

```
Query *pquery = retrieveQuery();
```

то была бы вызвана реализация print() из класса NameQuery. Обращение

```
cout << *pquery << endl;
```

приводит к вызову той функции print(), которая ассоциирована с объектом, адресуемым указателем pquery в данной точке выполнения программы.

17.5.2. Чисто виртуальные функции

С точки зрения кодирования основная задача, стоящая перед нами в связи с поддержкой пользовательских запросов, – это реализация зависимых от типа операций для каждого из возможных операторов. Для этого мы определили четыре конкретных типа классов: AndQuery, OrQuery и т.д. Однако с точки зрения проектирования наша цель – инкапсулировать обработку каждого вида запроса, спрятать за не зависящим от типа интерфейсом. Это позволит построить ядро приложения, которое не потребует изменений при добавлении или удалении типов.

Чтобы добиться этого, определим абстрактный тип класса Query. При этом мы не будем программировать разные типы пользовательских запросов, а лишь абстрактные

```
void doit_and_bedone( vector< Query* > *pvec )
{
   vector<Query*>::iterator
      it = pvec->begin(),
      end_it = pvec->end();

   for ( ; it != end_it; ++it )
   {
       Query *pq = *it;
       cout << "обрабатывается " << *pq << endl;
       pq->eval();
       pq->display();
       delete pq;
   }
```

операции, применимые к ним:

```
}
```

Такое определение позволяет добавлять неограниченное число типов запросов без необходимости изменять или даже перекомпилировать ядро системы, но при условии, что открытый интерфейс нашего абстрактного базового класса Query достаточен для поддержки новых запросов.

Проектируя открытый интерфейс Query, мы определим множество операций, достаточное для поддержки всех существующих и будущих типов запросов, хотя на практике нам вряд ли удастся это гарантировать. Предоставление общего интерфейса для тех запросов, о которых мы уже знаем, – вполне реальная задача, но любое заявление, претендующее на более широкую поддержку, следует рассматривать с долей скептицизма.

Поскольку Query – абстрактный класс, объекты которого в приложении не создаются, то никакой разумной реализации виртуальных функций в нем самом мы предложить не можем. Это лишь названия, которые должны быть замещены в производных классах. Напрямую вызывать их мы не будем.

Язык обладает синтаксической конструкцией, обозначающей, что некоторая виртуальная функция предоставляет интерфейс, который должен быть замещен в производных подтипах, но вызываться непосредственно не может. Это *чисто виртуальные функции*.

```
class Query {
public:
   // объявляется чисто виртуальная функция
   virtual ostream& print( ostream&=cout ) const = 0;
   // ...
};
```

Объявляются они следующим образом:

Заметьте, что за объявлением функции следует присваивание нуля.

Класс, содержащий (или наследующий) одну или несколько таких функций, распознается компилятором как абстрактный базовый класс. Попытка создать независимый объект абстрактного класса приводит к ошибке компиляции. (Ошибкой является также вызов

```
// В классе Query объявлены одна или несколько виртуальных функций,
// поэтому программист не может создавать независимые объекты
// класса Query

// правильно: подобъект Query в составе NameQuery
Query *pq = new NameQuery( "Nostromo" );

// ошибка: оператор new создает объект класса Query
```

чисто виртуальной функции с помощью механизма виртуализации.) Например:

```
Query *pq2 = new Query;
```

Абстрактный базовый класс может существовать только как подобъект в составе объекта некоторого производного от него класса. Это именно та семантика, которая нужна нам для базового Query.

17.5.3. Статический вызов виртуальной функции

Вызывая виртуальную функцию с помощью оператора разрешения области видимости класса, мы отменяем механизм виртуализации и разрешаем вызов статически, на этапе компиляции. Предположим, что мы определили виртуальную функцию `isA()` в базовом

```
Query *pquery = new NameQuery( "dumbo" );

// isA() вызывается динамически с помощью механизма виртуализации
// реально будет вызвана NameQuery::isA()
pquery->isA();

// isA вызывается статически во время компиляции
// реально будет вызвана Query::isA
```

и каждом из производных классов иерархии `Query`:

```
pquery->Query::isA();
```

Тогда явный вызов `Query::isA()` разрешается на этапе компиляции в пользу реализации `isA()` в базовом классе `Query`, хотя `pquery` адресует объект `NameQuery`.

Зачем нужно отменять механизм виртуализации? Как правило, ради эффективности. В теле виртуальной функции производного класса часто необходимо вызвать реализацию из базового, чтобы завершить операцию, расщепленную между базовым и производным классами. К примеру, вполне вероятно, что виртуальная функция `display()` из `Camera` выводит некоторую информацию, общую для всех камер, а реализация `display()` в классе `PerspectiveCamera` сообщает информацию, специфичную только для перспективных камер. Вместо того чтобы дублировать в ней действия, общие для всех камер, можно вызвать реализацию из класса `Camera`. Мы точно знаем, какая именно реализация нам нужна, поэтому нет нужды прибегать к механизму виртуализации. Более того, реализация в `Camera` объявлена встроенной, так что разрешение во время компиляции приводит к подстановке по месту вызова.

Приведем еще один пример, когда отмена механизма виртуализации может оказаться полезной, а заодно познакомимся с неким аспектом чисто виртуальных функций, который начинающим программистам кажется противоречащим интуиции.

Реализации функции `print()` в классах `AndQuery` и `OrQuery` совпадают во всем, кроме литеральной строки, представляющей название оператора. Реализуем только одну функцию, которую можно вызывать из данных классов. Для этого мы снова определим абстрактный базовый `BinaryQuery` (его наследники – `AndQuery` и `OrQuery`). В нем определены два операнда и еще один член типа `string` для хранения значения оператора. Поскольку это абстрактный класс, объявим `print()` чисто виртуальной функцией:

```
class BinaryQuery : public Query {
public:
   BinaryQuery( Query *lop, Query *rop, string oper )
              : _lop(lop), _rop(rop), _oper(oper) {}

   ~BinaryQuery() { delete _lop; delete _rop; }
   ostream &print( ostream&=cout, ) const = 0;

protected:
   Query *_lop;
   Query *_rop;
   string _oper;

};
```

Вот как реализована в BinaryQuery функция print(), которая будет вызываться из

```
inline ostream&
BinaryQuery::
print( ostream &os ) const
{
   if ( _lparen )
           print_lparen( _lparen, os );

   _lop->print( os );
   os << ' ' << _oper << ' ';
   _rop->print( os );

   if ( _rparen )
           print_rparen( _rparen, os );

   return os;
}
```

производных классов AndQuery и OrQuery:

Похоже, мы попали в парадоксальную ситуацию. С одной стороны, необходимо объявить этот экземпляр print() как чисто виртуальную функцию, чтобы компилятор воспринимал BinaryQuery как абстрактный базовый класс. Тогда в приложении определить независимые объекты BinaryQuery будет невозможно.

С другой стороны, нужно определить в классе BinaryQuery виртуальную функцию print() и уметь вызывать ее через объекты AndQuery и OrQuery.

Но как часто бывает с кажущимися парадоксами, мы не учли одного обстоятельства: чисто виртуальную функцию нельзя вызывать с помощью механизма виртуализации, но

```
inline ostream&
AndQuery::
print( ostream &os ) const
{
   // правильно: подавить механизм виртуализации
   // вызвать BinaryQuery::print статически
   BinaryQuery::print( os );
}
```

можно вызывать статически:

17.5.4. Виртуальные функции и аргументы по умолчанию

```
#include <iostream>

class base {
public:
   virtual int foo( int ival = 1024 ) {
      cout << "base::foo() -- ival: " << ival << endl;
      return ival;
   }

   // ...
};

class derived : public base {
public:
   virtual int foo( int ival = 2048 ) {
      cout << "derived::foo() -- ival: " << ival << endl;
      return ival;
   }

   // ...
};
```

Рассмотрим следующую простую иерархию классов:

Проектировщик класса хотел, чтобы при вызове без параметров реализации `foo()` из

```
base b;
base *pb = &b;

// вызывается base::foo( int )
// предполагалось, что будет возвращено 1024
```

базового класса по умолчанию передавался аргумент 1024:

```
pb->foo();
```

Кроме того, разработчик хотел, чтобы при вызове его реализации `foo()` без параметров

```
derived d;
base *pb = &d;

// вызывается derived::foo( int )
// предполагалось, что будет возвращено 2048
```

использовался аргумент по умолчанию 2048:

```
pb->foo();
```

Однако в C++ принята другая семантика механизма виртуализации. Вот небольшая программа для тестирования нашей иерархии классов:

```
int main()
{
   derived *pd = new derived;
   base *pb = pd;

   int val = pb->foo();
   cout << "main() : val через base: "
        << val << endl;

   val = pd->foo();
   cout << "main() : val через derived: "
        << val << endl;
}
```

После компиляции и запуска программа выводит следующую информацию:

```
derived::foo() -- ival: 1024
main() : val через base: 1024
derived::foo() -- ival: 2048
main() : val через derived: 2048
```

При обоих обращениях реализация `foo()` из производного класса вызывается корректно, поскольку фактически вызываемый экземпляр определяется во время выполнения на основе типа класса, адресуемого `pd` и `pb`. Но передаваемый `foo()` аргумент по умолчанию определяется не во время выполнения, а во время компиляции на основе типа объекта, через который вызывается функция. При вызове `foo()` через `pb` аргумент по умолчанию извлекается из объявления `base::foo()` и равен 1024. Если же `foo()` вызывается через `pd`, то аргумент по умолчанию извлекается из объявления `derived::foo()` и равен 2048.

Если реализации из производного класса при вызове через указатель или ссылку на базовый класс по умолчанию передается аргумент, указанный в базовом классе, то зачем задавать аргумент по умолчанию для реализации из производного класса?

Нам могут понадобиться различные аргументы по умолчанию в зависимости не от реализации `foo()` в конкретном производном классе, а от типа указателя или ссылки, через которые функция вызвана. Например, значения 1024 и 2048 – это размеры изображений. Когда нужно получить менее детальное изображение, вызываем `foo()` через класс `base`, а когда более детальное – через `derived`.

Но если мы все-таки хотим, чтобы аргумент по умолчанию, передаваемый `foo()`, зависел от фактически вызванного экземпляра? К сожалению, механизм виртуализации такую возможность не поддерживает. Однако разрешается задать такой аргумент по умолчанию, который для вызванной функции означает, что пользователь не передал никакого значения. Тогда реальное значение, которое функция хотела бы видеть в качестве аргумента по умолчанию, объявляется локальной переменной и используется, если ничего другого не передано:

```
void
base::
foo( int ival = base_default_value )
{
   int real_default_value = 1024;     // настоящее значение по умолчанию

   if ( ival == base_default_value )
      ival = real_default_value;

   // ...

}
```

Здесь base_default_value – значение, согласованное между всеми классами иерархии, которое явно говорит о том, что пользователь не передал никакого аргумента.

```
void
derived::
foo( int ival = base_default_value )
{
   int real_default_value = 2048;

   if ( ival == base_default_value )
      ival = real_default_value;

   // ...

}
```

Производный класс может быть реализован аналогично:

17.5.5. Виртуальные деструкторы

```
void doit_and_bedone( vector< Query* > *pvec )
{
   // ...
   for ( ; it != end_it; ++it )
   {
       Query *pq = *it;
       // ...
       delete pq;
   }

}
```

В данной функции мы применяем оператор delete:

Чтобы функция выполнялась правильно, применение delete должно вызывать деструктор того класса, на который указывает pq. Следовательно, необходимо объявить деструктор Query виртуальным:

```
class Query {
public:
   virtual ~Query() { delete _solution; }
   // ...
};
```

Деструкторы всех производных от `Query` классов автоматически считаются виртуальными. `doit_and_bedone()` выполняется правильно.

Поведение деструктора при наследовании таково: сначала вызывается деструктор производного класса, в случае `pq` – виртуальная функция. По завершении вызывается деструктор непосредственного базового класса – статически. Если деструктор объявлен встроенным, то в точке вызова производится подстановка. Например, если `pq` указывает на объект класса `AndQuery`, то

```
delete pq;
```

приводит к вызову деструктора класса `AndQuery` за счет механизма виртуализации. После этого статически вызывается деструктор `BinaryObject`, а затем – снова статически – деструктор `Query`.

```
class Query {
public:  // ...
protected:
   virtual ~Query();
   // ...
};

class NotQuery : public Query {
public:
   ~NotQuery();
   // ...
};
```

В следующей иерархии классов

уровень доступа к конструктору `NotQuery` открытый при вызове через объект `NotQuery`, но защищенный – при вызове через указатель или ссылку на объект `Query`. Таким образом, виртуальная функция подразумевает уровень доступа того класса, через объект

```
int main()
{
   Query *pq = new NotQuery;

   // ошибка: деструктор является защищенным
   delete pq;
}
```

которого вызывается:

Эвристическое правило: если в корневом базовом классе иерархии объявлены одна или несколько виртуальных функций, рекомендуем объявлять таковым и деструктор. Однако, в отличие от конструктора базового класса, его деструктор не стоит делать защищенным.

17.5.6. Виртуальная функция eval()

В основе иерархии классов `Query` лежит виртуальная функция `eval()` (но с точки зрения возможностей языка она наименее интересна). Как и для других функций-членов, разумной реализации `eval()` в абстрактном классе `Query` нет, поэтому мы объявляем ее

```
class Query {
public:
   virtual void eval() = 0;
   // ...
```

чисто виртуальной:

```
};
```

Реальное разрешение имени `eval()` происходит при построении отображения слов на вектор позиций. Если слово есть в тексте, то в отображении будет его вектор позиций. В нашей реализации вектор позиций, если он имеется, передается конструктору `NameQuery` вместе с самим словом. Поэтому в классе `NameQuery` функция `eval()` пуста.

Однако мы не можем унаследовать чисто виртуальную функцию из `Query`. Почему? Потому что `NameQuery` – это конкретный класс, объекты которого разрешается создавать в приложении. Если бы мы унаследовали чисто виртуальную функцию, то он стал бы абстрактным классом, так что создать объект такого типа не удалось бы. Поэтому мы

```
class NameQuery : public Query {
public:
   virtual void eval() {}
   // ...
```

объявим `eval()` пустой функцией:

```
};
```

Для запроса `NotQuery` отыскиваются все строки текста, где указанное слово отсутствует. Для таких строк в член `_loc` класса `NotQuery` помещаются все пары (строка, колонка). Наша реализация выглядит следующим образом:

```
void NotQuery::eval()
{
    // вычислим операнд
    _op->eval();

    // _all_locs - это вектор, содержащий начальные позиции всех слов,
    // он является статическим членом NotQuery:
    // static const vector<locations>* _all_locs
    vector< location >::const_iterator
            iter = _all_locs->begin(),
            iter_end = _all_locs->end();

    // получить множество строк, в которых операнд встречается
    set<short> *ps = _vec2set( _op->locations() );

    // для каждой строки, где операнд не найден,
    // скопировать все позиции в _loc
    for ( ; iter != iter_end; ++iter )
    {
            if ( ! ps->count( (*iter).first )) {
           _loc.push_back( *iter );
            }
    }
}
```

Ниже приводится трассировка выполнения запроса `NotQuery`. Операнд встречается в 0, 3 и 5 строках текста. (Напомним, что внутри программы строки текста в векторе нумеруются с 0; а когда мы предъявляем строки пользователю, мы нумеруем их с единицы.) Поэтому при вычислении ответа создается вектор, содержащий начальные позиции слов в строках 1,2 и 4. (Мы отредактировали вектор позиций, чтобы он занимал меньше места.)

```
==> ! daddy
daddy ( 3 ) lines match
display_location_vector:
        first: 0          second: 8
        first: 3          second: 3
        first: 5          second: 5
! daddy ( 3 ) lines match
display_location_vector:
        first: 1          second: 0
        first: 1          second: 1
        first: 1          second: 2
        ...
        first: 1          second: 10
        first: 2          second: 0
        first: 2          second: 1
        ...
        first: 2          second: 12
        first: 4          second: 0
        first: 4          second: 1
        ...
        first: 4          second: 12

Requested query:    ! daddy
( 2 ) when the wind blows through her hair, it looks almost alive,
( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
( 5 ) she tells him, at the same time wanting him to tell her more.
```

При обработке запроса `OrQuery` векторы позиций обоих операндов объединяются. Для этого применяется обобщенный алгоритм `merge()`. Чтобы `merge()` мог упорядочить

пары (строка, колонка), мы определяем объект-функцию для их сравнения. Ниже

```
class less_than_pair {
public:
          bool operator()( location loc1, location loc2 )
          {
          return (( loc1.first < loc2.first ) ||
                  ( loc1.first == loc2.first ) &&
                  ( loc1.second < loc2.second ));
          }
};
void OrQuery::eval()
{
    // вычислить левый и правый операнды
    _lop->eval();
    _rop->eval();

    // подготовиться к объединению двух векторов позиций
    vector< location, allocator >::const_iterator
        riter = _rop->locations()->begin(),
        liter = _lop->locations()->begin(),
        riter_end = _rop->locations()->end(),
        liter_end = _lop->locations()->end();

    merge( liter, liter_end, riter, riter_end,
               inserter( _loc, _loc.begin() ),
               less_than_pair() );
```

приведена наша реализация:

```
}
```

А вот трассировка выполнения запроса `OrQuery`, в которой мы выводим вектор позиций каждого из двух операндов и результат их объединения. (Напомним еще раз, что для пользователя строки нумеруются с 1, а внутри программы – с 0.)

```
==> fiery || untamed
fiery ( 1 ) lines match
display_location vector:
        first: 2          second: 2
        first: 2          second: 8

untamed ( 1 ) lines match
display_location vector:
        first: 3          second: 2

fiery || untamed ( 2 ) lines match
display_location vector:
        first: 2          second: 2
        first: 2          second: 8
        first: 3          second: 2

Requested query: fiery || untamed
( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
( 4 ) magical but untamed. "Daddy, shush, there is no such thing,"
```

При обработке запроса `AndQuery` мы обходим векторы позиций обоих операндов и ищем соседние слова. Каждая найденная пара вставляется в вектор `_loc`. Основная трудность связана с тем, что эти векторы нужно просматривать синхронно, чтобы можно было установить соседство слов.

```
void AndQuery::eval()
{
    // вычислить левый и правый операнды
    _lop->eval();
    _rop->eval();

    // установить итераторы
    vector< location, allocator >::const_iterator
        riter = _rop->locations()->begin(),
        liter = _lop->locations()->begin(),
        riter_end = _rop->locations()->end(),
        liter_end = _lop->locations()->end();

    // продолжать цикл, пока есть что сравнивать
    while ( liter != liter_end &&
                riter != riter_end )
    {
            // пока номер строки в левом векторе больше, чем в правом
            while ( (*liter).first > (*riter).first )
            {
               ++riter;
               if ( riter == riter_end ) return;
            }

            // пока номер строки в левом векторе меньше, чем в правом
            while ( (*liter).first < (*riter).first )
            {
          // если соответствие найдено для последнего слова
          // в одной строке и первого слова в следующей
          // _max_col идентифицирует последнее слово в строке
               if ( ((*liter).first == (*riter).first-1 ) &&
              ((*riter).second == 0 ) &&
              ((*liter).second == (*_max_col)[ (*liter).first ] ))
               {
            _loc.push_back( *liter );
            _loc.push_back( *riter );
            ++riter;
            if ( riter == riter_end ) return;
               }
               ++liter;
               if ( liter == liter_end ) return;
      }

      // пока оба в одной и той же строке
      while ( (*liter).first == (*riter).first )
      {
         if ( (*liter).second+1 == ((*riter).second) )
         {  // соседние слова
            _loc.push_back( *liter ); ++liter;
            _loc.push_back( *riter ); ++riter;
         }
         else
         if ( (*liter).second <= (*riter).second )
            ++liter;
         else ++riter;
         if ( liter == liter_end || riter == riter_end )
                 return;
      }
    }
}
```

А так выглядит трассировка выполнения запроса AndQuery, в которой мы выводим векторы позиций обоих операндов и результирующий вектор:

```
==> fiery && bird
fiery ( 1 ) lines match
display_location vector:
        first: 2          second: 2
        first: 2          second: 8
bird ( 1 ) lines match
display_location vector:
        first: 2          second: 3
        first: 2          second: 9
fiery && bird ( 1 ) lines match
display_location vector:
        first: 2          second: 2
        first: 2          second: 3
        first: 2          second: 8
        first: 2          second: 9

Requested query: fiery && bird
( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
```

Приведем трассировку выполнения составного запроса, включающего как И, так и ИЛИ. Показаны векторы позиций каждого операнда, а также результирующий вектор:

```
==> fiery && ( bird || untamed )
fiery ( 1 ) lines match
display_location vector:
        first: 2          second: 3
        first: 2          second: 8
bird ( 1 ) lines match
display_location vector:
        first: 2          second: 3
        first: 2          second: 9
untamed ( 1 ) lines match
display_location vector:
        first: 3         second: 2
( bird || untamed ) ( 2 ) lines match
display_location vector:
        first: 2          second: 3
        first: 2          second: 9
        first: 3          second: 2
fiery && ( bird || untamed ) ( 1 ) lines match
display_location vector:
        first: 2          second: 2
        first: 2          second: 3
        first: 2          second: 8
        first: 2          second: 9
Requested query: fiery && ( bird || untamed )
( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
```

17.5.7. Почти виртуальный оператор new

Если дан указатель на один из конкретных подтипов запроса, то разместить в хипе

```
NotQuery *pnq;
// установить pnq ...

// оператор new вызывает
// копирующий конструктор NotQuery ...
```

дубликат объекта несложно:

```
NotQuery *pnq2 = new NotQuery( *pnq );
```

Если же у нас есть только указатель на абстрактный класс Query, то задача создания

```
const Query *pq = pnq->op();
```

дубликата становится куда менее тривиальной:

```
// как получить дубликат pq?
```

Если бы позволялось объявить виртуальный экземпляр оператора new, то проблема была бы решена, поскольку автоматически вызывался бы нужный экземпляр. К сожалению, это невозможно: new – статическая функция-член, которая применяется к неструктурированной памяти еще до конструирования объекта класса (см. раздел 15.8).

Но хотя оператор new нельзя сделать виртуальным, разрешается создать его суррогат,

```
class Query {
public:
   virtual Query *clone() = 0;
   // ...
```

который будет выделять память из хипа и копировать туда объекты, – clone():

```
};
```

```
class NameQuery : public Query {
public:
   virtual Query *clone()
      // вызывается копирующий конструктор класса NameQuery
      { return new NameQuery( *this ); }

   // ...
```

Вот как он может быть реализован в классе NameQuery:

```
};
```

```
Query *pq = new NameQuery( "valery" );
```

Это работает правильно, если тип целевого указателя Query*:

```
Query *pq2 = pq->clone();
```

Если же его тип равен NameQuery*, нужно привести возвращенный указатель типа

```
NameQuery *pnq = new NameQuery( "Rilke" );
NameQuery *pnq2 =
```

Query* назад к типу NameQuery*:

```
    static_cast<NameQuery*>( pnq->clone() );
```

(Причина, по которой необходимо преобразование типа, объясняется в разделе 19.1.1.)

Как правило, тип значения, возвращаемого реализацией виртуальной функции в производном классе, должен совпадать с типом, возвращаемым ее реализацией в базовом. Исключение, о котором мы уже упоминали, призвано поддержать рассмотренную ситуацию. Если виртуальная функция в базовом классе возвращает значение некоторого типа класса (либо указатель или ссылку на тип класса), то ее реализация в производном может возвращать значение, тип которого является производным от этого класса с

```
class NameQuery : public Query {
public:
   virtual NameQuery *clone()
      { return new NameQuery( *this ); }

// ...
```

открытым типом наследования (то же относится к ссылкам и указателям):

```
};
```

```
// Query *pq = new NameQuery( "Broch" );
Query *pq2 = pq->clone();   // правильно

// NameQuery *pnq = new NameQuery( "Rilke" );
```

Теперь pq2 и pnq2 можно инициализировать без явного приведения типов:

```
NameQuery *pnq2 = pnq->clone();   // правильно
```

```
class NotQuery : public Query {
public:
   virtual NotQuery *clone()
      { return new NotQuery( *this ); }

   // ...
```

Так выглядит реализация clone() в классе NotQuery:

```
};
```

Реализации в AndQuery и OrQuery аналогичны. Чтобы эти реализации clone() работали правильно, в классах NotQuery, AndQuery и OrQuery должны быть явно определены копирующие конструкторы. (Мы займемся этим в разделе 17.6.)

17.5.8. Виртуальные функции, конструкторы и деструкторы

Как мы видели в разделе 17.4, для объекта производного класса сначала вызывается конструктор базового, а затем производного класса. Например, при таком определении объекта NameQuery

```
NameQuery poet( "Orlen" );
```

сначала будет вызван конструктор Query, а потом NameQuery.

При выполнении конструктора базового класса `Query` часть объекта, соответствующая классу `NameQuery`, остается неинициализированной. По существу, `poet` – это еще не объект `NameQuery`, сконструирован лишь его подобъект.

Что должно происходить, если внутри конструктора базового класса вызывается виртуальная функция, реализации которой существуют как в базовом, так и в производном классах? Какая из них должна быть вызвана? Результат вызова реализации из производного класса в случае, когда необходим доступ к его членам, оказался бы неопределенным. Вероятно, выполнение программы закончилось бы крахом.

Чтобы этого не случилось, в конструкторе базового класса всегда вызывается реализация виртуальной функции, определенная именно в базовом. Иными словами, внутри такого конструктора объект производного класса рассматривается как имеющий тип базового.

То же самое справедливо и внутри деструктора базового класса, вызываемого для объекта производного. И в этом случае часть объекта, относящаяся к производному классу, не определена: не потому, что еще не сконструирована, а потому, что уже уничтожена.

Упражнение 17.12

Внутри объекта `NameQuery` естественное внутреннее представление вектора позиций – это указатель, который инициализируется указателем, хранящимся в отображении слов. Оно же является и наиболее эффективным, так как нам нужно скопировать лишь один адрес, а не каждую пару координат. Классы `AndQuery`, `OrQuery` и `NotQuery` должны конструировать собственные векторы позиций на основе вычисления своих операндов. Когда время жизни объекта любого из этих классов завершается, ассоциированный с ним вектор позиций необходимо удалить. Когда же заканчивается время жизни объекта `NameQuery`, вектор позиций удалять *не следует*. Как сделать так, чтобы вектор позиций был представлен указателем в базовом классе `Query` и при этом его экземпляры для объектов `AndQuery`, `OrQuery` и `NotQuery` удалялись, а для объектов `NameQuery` – нет? (Заметим, что нам не разрешается добавить в класс `Query` признак, показывающий, нужно ли применять оператор `delete` к вектору позиций!)

Упражнение 17.13

```
class AbstractObject {
public:
   ~AbstractObject();
   virtual void doit() = 0;
   // ...
```

Что неправильно в приведенном определении класса:

```
};
```

Упражнение 17.14

```
NameQuery nq( "Sneezy" );
Query q( nq );
```

Даны такие определения:

```
Query *pq = &nq;
```

Почему в инструкции

```
pq->eval();
```

вызывается экземпляр `eval()` из класса `NameQuery`, а в инструкции

```
q.eval();
```

экземпляр из `Query`?

Упражнение 17.15

```
(a) Base* Base::copy( Base* );
```

Какие из повторных объявлений виртуальных функций в классе `Derived` неправильны:

```
(b) Base* Base::copy( Base* );
    Base* Derived::copy( Derived* );
```

```
(c) ostream& Base::print( int, ostream&=cout );
    Derived* Derived::copy( Vase* );
```

```
(d) void Base::eval() const;
    ostream& Derived::print( int, ostream& );
    void Derived::eval();
```

Упражнение 17.16

Маловероятно, что наша программа заработает при первом же запуске и в первый раз, когда прогоняется с реальными данными. Средства отладки полезно включать уже на этапе проектирования классов. Реализуйте в нашей иерархии классов `Query` виртуальную функцию `debug()`, которая будет отображать члены соответствующих классов. Поддержите управление уровнем детализации двумя способами: с помощью аргумента, передаваемого функции `debug()`, и с помощью члена класса. (Последнее позволяет включать или отключать выдачу отладочной информации в отдельных объектах.)

Упражнение 17.17

Найдите ошибку в следующей иерархии классов:

```
class Object {
public:
   virtual void doit() = 0;
   // ...
protected:
   virtual ~Object();
};

class MyObject : public Object {
public:
   MyObject( string isA );
   string isA() const;
protected:
   string _isA;
};
```

17.6. Почленная инициализация и присваивание A

При проектировании класса мы должны позаботиться о том, чтобы почленная инициализация (см. раздел 14.6) и почленное присваивание (см. раздел 14.7) были реализованы правильно и эффективно. Рассмотрим связь этих операций с наследованием.

До сих пор мы не занимались явной обработкой почленной инициализации. Посмотрим, что происходит в нашей иерархии классов `Query` по умолчанию.

```
class Query {
public: // ...
protected:
   int _paren;
   set<short> *_solition;
   vector<location> _loc;
   // ...
};
```

В абстрактном базовом классе `Query` определены три нестатических члена:

Член `_solution`, если он установлен, адресует множество, память для которого выделена в хипе функцией-членом `_vec2set()`. Деструктор `Query` применяет к `_solution` оператор `delete`.

Класс `Query` должен предоставлять как явный копирующий конструктор, так и явный копирующий оператор присваивания. (Если вам это непонятно, перечитайте раздел 14.6.) Но сначала посмотрим, как почленное копирование по умолчанию происходит без них.

Производный класс `NameQuery` содержит объект-член типа `string` и подобъект базового `Query`. Если есть объект `folk` класса `NameQuery`:

```
NameQuery folk( "folk" );
```

то инициализация `music` с помощью `folk`

```
NameQuery music = folk;
```

осуществляется так:

1. Компилятор проверяет, есть ли в NameQuery явный копирующий конструктор. (Его нет. Поэтому необходимо применить почленную инициализацию по умолчанию.)
2. Далее компилятор проверяет, содержит ли объект NameQuery подобъекты базового класса. (Да, в нем имеется подобъект Query.)
3. Компилятор проверяет, определен ли в классе Query явный копирующий конструктор. (Нет, поэтому компилятор применит почленную инициализацию по умолчанию.)
4. Компилятор проверяет, содержит ли объект Query подобъекты базового класса. (Нет.)
5. Компилятор просматривает все нестатические члены Query в порядке их объявления. (Если некоторый член не является объектом класса, как, например, _paren и _solution, то в объекте music он инициализируется соответствующим членом объекта folk. Если же является, как, скажем, _loc, то к нему рекурсивно применяется шаг 1. В классе vector определен копирующий конструктор, который вызывается для инициализации music._loc с помощью folk._loc.)
6. Далее компилятор рассматривает нестатические члены NameQuery в порядке их объявления и находит объект класса string, где есть явный копирующий конструктор. Он и вызывается для инициализации music._name с помощью folk._name.

Инициализация по умолчанию music с помощью folk завершена. Она хороша во всех отношениях, кроме одного: если разрешить копирование по умолчанию члена _solution, то программа, скорее всего, завершится аварийно. Поэтому вместо такой обработки мы предоставим явный копирующий конструктор класса Query. Можно, например,

```
Query::Query( const Query &rhs )
            : _loc( rhs._loc ), _paren(rhs._paren)
{
   if ( rhs._solution )
   {
      _solution = new set<short>;
      set<short>::iterator
                  it = rhs._solution->begin(),
                  end_it = rhs._solution->end();

      for ( ; _ir != end_it; ++it )
          _solution->insert( *it );
   }
   else _solution = 0;
```

скопировать все разрешающее множество:

```
}
```

Однако, поскольку в нашей реализации разрешающее множество вычисляется по мере необходимости, копировать его сразу нет нужды. Назначение нашего копирующего конструктора – предотвратить копирование по умолчанию. Для этого достаточно инициализировать _solution нулем:

```
Query::Query( const Query &rhs )
            : _loc( rhs._loc ),
              _paren(rhs._paren), _solution( 0 )
{}
```

Шаги 1 и 2 инициализации `music` с помощью `folk` те же, что и раньше. Но на шаге 3 компилятор обнаруживает, что в классе `Query` есть явный копирующий конструктор и вызывает его. Шаги 4 и 5 пропускаются, а шаг 6 выполняется, как и прежде.

На этот раз почленная инициализация `music` с помощью `folk` корректна. Реализовывать явный копирующий конструктор в `NameQuery` нет необходимости.

Объект производного класса `NotQuery` содержит подобъект базового `Query` и член `_op` типа `Query*`, который указывает на операнд, размещенный в хипе. Деструктор `NotQuery` применяет к этому операнду оператор `delete`.

Для класса `NotQuery` почленная инициализация по умолчанию члена `_op` небезопасна, поэтому необходим явный копирующий конструктор. В его реализации используется

```
inline NotQuery::
NotQuery( const NotQuery &rhs )
        // вызывается Query::Query( const Query &rhs )
        : Query( rhs )
```

виртуальная функция `clone()`, которую мы определили в предыдущем разделе.

```
        { _op = rhs._op->clone(); }
```

При почленной инициализации одного объекта класса `NotQuery` другим выполняются два шага:

1. Компилятор проверяет, определен ли в `NotQuery` явный копирующий конструктор. Да, определен.
2. Этот конструктор вызывается для почленной инициализации.

Вот и все. Ответственность за правильную инициализацию подобъекта базового класса и нестатических членов возлагается на копирующий конструктор `NotQuery`. (Классы `AndQuery` и `OrQuery` сходны с `NotQuery`, поэтому мы оставляем их в качестве упражнения для читателей.)

Почленное присваивание аналогично почленной инициализации. Если имеется явный копирующий оператор присваивания, то он вызывается для выполнения присваивания одного объекта класса другому. В противном случае применяется почленное присваивание по умолчанию.

Если базовый класс есть, то сначала с помощью копирующего оператора присваивания почленно присваивается подобъект данного класса, иначе такое присваивание рекурсивно применяется к базовым классам и членам подобъекта базового класса.

Просматриваются все нестатические члены в порядке их объявления. Если член не является объектом класса, то его значение справа от знака равенства копируется в значение соответствующего члена слева от знака равенства. Если же член является объектом класса, в котором определен явный копирующий оператор присваивания, то он и вызывается. В противном случае к базовым классам и членам объекта-члена применяется почленное присваивание по умолчанию.

Вот как выглядит копирующий оператор присваивания для нашего объекта Query. Еще раз отметим, что в этом месте необязательно копировать разрешающее множество,

```
Query&
Query::
operator=( const Query &rhs )
{
   // предотвратить присваивание самому себе
   if ( &rhs != this )
   {
      _paren = rhs._paren;
      _loc = rhs._loc;
      delete _solution;
      _solution = 0;
   }

   return *this;
```

достаточно предотвратить копирование по умолчанию:

```
};
```

В классе NameQuery явный копирующий оператор присваивания не нужен. Присваивание одного объекта NameQuery другому выполняется в два шага:

1. Для присваивания подобъектов Query двух объектов NameQuery вызывается явный копирующий оператор присваивания класса Query.
2. Для присваивания членов string вызывается явный копирующий оператор присваивания этого класса.

Для объектов NameQuery вполне достаточно почленного присваивания по умолчанию.

В каждом из классов NotQuery, AndQuery и OrQuery для безопасного копирования операндов требуется явный копирующий оператор присваивания. Вот его реализация для

```
inline NotQuery&
NotQuery::
operator=( const NotQuery &rhs )
{
   // предотвратить присваивание самому себе
   if ( &rhs != this )
   {
       // вызвать копирующий оператор присваивания Query
            this->Query::operator=( rhs );

       // скопировать операнд
            _op = rhs._op->clone();
   }

   return *this;
```

NotQuery:

```
}
```

В отличие от копирующего конструктора, в копирующем операторе присваивания нет специальной части, через которую вызывается аналогичный оператор базового класса. Для этого используются две синтаксических конструкции: явный вызов, продемонстрированный выше, и явное приведение типа, как в следующем примере:

```
(*static_cast<Query*>(this)) = rhs;
```

(Реализация копирующих операторов присваивания в классах AndQuery и OrQuery выглядит так же, поэтому мы оставим ее в качестве упражнения.)

Ниже предложена небольшая программа для тестирования данной реализации. Мы

```
#include "Query.h"

int
main()
{
   NameQuery nm( "alice" );
   NameQuery nm( "emma" );

   NotQuery nq1( &nm );
   cout << "notQuery 1: " << nq1 << endl;

   NotQuery nq2( nq1 );
   cout << "notQuery 2: " << nq2 << endl;
   NotQuery nq3( &nm2 );
   cout << "notQuery 3: " << nq3 << endl;

   nq3 = nq2;
   cout << "notQuery 3 присвоено значение nq2: " << nq3 << endl;

   AndQuery aq( &nq1, &nm2 );
   cout << "AndQuery : " << aq << endl;

   AndQuery aq2( aq );
   cout << "AndQuery 2: " << aq2 << endl;

   AndQuery aq3( &nm, &nm2 );
   cout << "AndQuery 3: " << aq3 << endl;

   aq2 = aq3;
   cout << "AndQuery 2 после присваивания: " << aq2 << endl;
```

создаем или копируем объект, а затем распечатываем его.

```
}
```

После компиляции и запуска программа печатает следующее:

```
notQuery 1:  ! alice
notQuery 2:  ! alice
notQuery 3:  ! emma
notQuery 3 присвоено значение nq2:  ! alice
AndQuery :  ! alice && emma
AndQuery 2:  ! alice && emma
AndQuery 3:  alice && emma
AndQuery 2 после присваивания:  alice && emma
```

Упражнение 17.18

Реализуйте копирующие конструкторы в классах AndQuery и OrQuery.

Упражнение 17.19

Реализуйте копирующие операторы присваивания в классах AndQuery и OrQuery.

Упражнение 17.20

Что указывает на необходимость реализации явных копирующего конструктора и копирующего оператора присваивания?

17.7. Управляющий класс UserQuery

Если имеется запрос такого типа:

```
fiery && ( bird || potato )
```

```
AndQuery
   NameQuery( "fiery" )
   OrQuery
      NameQuery( "bird" )
      NameQuery( "potato" )
```

то в нашу задачу входит построение эквивалентной иерархии классов:

Как лучше всего это сделать? Процедура вычисления ответа на запрос напоминает функционирование конечного автомата. Мы начинаем с пустого состояния и при обработке каждого элемента запроса переходим в новое состояние, пока весь запрос не будет разобран. В основе нашей реализации лежит одна инструкция `switch` внутри операции, которую мы назвали `eval_query()`. Слова запроса считываются одно за другим из вектора строк и сравниваются с каждым из возможных значений:

```
vector<string>::iterator
    it     = _query->begin(),
    end_it = _query->end();

for ( ; it != end_it; ++it )
            switch( evalQueryString( *it ))
            {
               case WORD:
                    evalWord( *it );
              break;

               case AND:
              evalAnd();
              break;

               case OR:
              evalOr();
              break;

               case NOT:
              evalNot();
              break;

               case LPAREN:
              ++_paren;
              ++_lparenOn;
              break;

               case RPAREN:
              --_paren;
              ++_rparenOn;
              evalRParen();
              break;

            }
```

Пять операций `eval`: `evalWord()`, `evalAnd()`, `evalOr()`, `evalNot` и `evalRParen()` – как раз и строят иерархию классов `Query`. Прежде чем обратиться к деталям их реализации, рассмотрим общую организацию программы.

Нам нужно определить каждую операцию в виде отдельной функции, как это было сделано в главе 6 при построении процедур обработки запроса. Пользовательский запрос и производные от `Query` классы представляют независимые данные, которыми оперируют эти функции. От такой модели программирования (она называется процедурной) мы предпочли отказаться.

В разделе 6.14 мы ввели класс `TextQuery`, где инкапсулировали операции и данные, изучавшиеся в главе 6. Здесь нам потребуется класс `UserQuery`, решающий аналогичные задачи.

Одним из членов этого класса должен быть вектор строк, содержащий сам запрос пользователя. Другой член – это указатель типа `Query*` на иерархическое представление запроса, построенное в `eval_query()`. Еще три члена служат для обработки скобок:

- `_paren` помогает изменить подразумеваемый порядок вычисления операторов (чуть позже мы продемонстрируем это на примере);
- `_lparenOn` и `_rparenOn` содержат счетчики левых и правых скобок, ассоциированные с текущим узлом дерева разбора запроса (мы показывали, как они используются, при обсуждении виртуальной функции `print()` в разделе 17.5.1).

Помимо этих пяти членов, нам понадобятся еще два. Рассмотрим следующий запрос:

```
fiery || untamed
```

```
OrQuery
    NameQuery( "fiery" )
```

Наша цель – представить его в виде следующего объекта `OrQuery`:

```
    NameQuery( "untamed" )
```

Однако порядок обработки такого запроса вызывает некоторые проблемы. Когда мы определяем объект `NameQuery`, объект `OrQuery` , к которому его надо добавить, еще не определен. Поэтому необходимо место, где можно временно сохранить объект `NameQuery`.

Чтобы сохранить что-либо для последующего использования, традиционно применяется стек. Поместим туда наш объект `NameQuery`. А когда позже встретим оператор ИЛИ (объект `OrQuery`), то достанем `NameQuery` из стека и присоединим его к `OrQuery` в качестве левого операнда.

Объект `OrQuery` неполон: в нем не хватает правого операнда. До тех пор пока этот операнд не будет построен, работу с данным объектом придется прекратить.

Его можно поместить в тот же самый стек, что и `NameQuery`. Однако `OrQuery` представляет другое состояние обработки запроса: это неполный оператор. Поэтому мы определим два стека: `_query_stack` для хранения объектов, представляющих сконструированные операнды составного запроса (туда мы помещаем объект `NameQuery`), а второй для хранения неполных операторов с отсутствующим правым операндом. Второй стек можно трактовать как место для хранения текущей операции, подлежащей завершению, поэтому назовем его `_current_op`. Сюда мы и поместим объект `OrQuery`. После того как второй объект `NameQuery` будет определен, мы достанем объект `OrQuery` из стека `_current_op` и добавим к нему `NameQuery` в качестве правого операнда. Теперь объект `OrQuery` завершен и мы можем поместить его в стек `_query_stack`.

Если обработка запроса завершилась нормально, то стек `_current_op` пуст, а в стеке `_query_stack` содержится единственный объект, который и представляет весь пользовательский запрос. В нашем случае это объект класса `OrQuery`.

Рассмотрим несколько примеров. Первый из них – простой запрос типа `NotQuery`:

```
! daddy
```

Ниже показана трассировка его обработки. Финальным объектом в стеке `_query_stack` является объект класса `NotQuery`:

```
evalNot() : incomplete!
      push on _current_op ( size == 1 )
evalWord() : daddy
      pop _current_op : NotQuery
      add operand: WordQuery : NotQuery complete!

      push NotQuery on _query_stack
```

Текст, расположенный с отступом под функциями `eval`, показывает, как выполняется операция.

Во втором примере – составном запросе типа `OrQuery` – встречаются оба случая. Здесь же иллюстрируется помещение полного оператора в стек `_query_stack`:

```
==> fiery || untamed || shyly

evalWord() : fiery
      push word on _query_stack
evalOr() : incomplete!
      pop _query_stack : fiery
      add operand : WordQuery : OrQuery incomplete!
      push OrQuery on _current_op ( size == 1 )
evalWord() : untamed
      pop _current_op : OrQuery
      add operand : WordQuery : OrQuery complete!
      push OrQuery on _query_stack
evalOr() : incomplete!
      pop _query_stack : OrQuery
      add operand : OrQuery : OrQuery incomplete!
      push OrQuery on _current_op ( size == 1 )
evalWord() : shyly
      pop _current_op : OrQuery
      add operand : WordQuery : OrQuery complete!
      push OrQuery on _query_stack
```

В последнем примере рассматривается составной запрос и применение скобок для изменения порядка вычислений:

```
==> fiery && ( bird || untamed )

evalWord() : fiery
      push word on _query_stack
evalAnd() : incomplete!
      pop _query_stack : fiery
      add operand : WordQuery : AndQuery incomplete!
      push AndQuery on _current_op ( size == 1 )
evalWord() : bird
      _paren is set to 1
      push word on _query_stack
evalOr() : incomplete!
      pop _query_stack : bird
      add operand : WordQuery : OrQuery incomplete!
      push OrQuery on _current_op ( size == 2 )
evalWord() : untamed
      pop _current_op : OrQuery
      add operand : WordQuery : OrQuery complete!
      push OrQuery on _query_stack
evalRParen() :
      _paren: 0 _current_op.size(): 1
      pop _query_stack : OrQuery
      pop _current_op : AndQuery
      add operand : OrQuery : AndQuery complete!
      push AndQuery on _query_stack
```

Реализация системы текстового поиска состоит из трех компонентов:

- класс TextQuery, где производится обработка текста (подробно он рассматривался в разделе 16.4). Для него нет производных классов;
- объектно-ориентированная иерархия Query для представления и обработки различных типов запросов;
- класс UserQuery, с помощью которого представлен конечный автомат для построения иерархии Query.

До настоящего момента мы реализовали эти три компонента практически независимо друг от друга и без каких бы то ни было конфликтов. Но, к сожалению, иерархия классов Query не поддерживает требований к конструированию объектов, предъявляемых реализацией UserQuery:

- классы AndQuery, OrQuery и NotQuery требуют, чтобы каждый операнд присутствовал в момент определения объекта. Однако принятая нами схема обработки подразумевает наличие неполных объектов;
- наша схема предполагает отложенное добавление операнда к объектам AndQuery, OrQuery и NotQuery. Более того, такая операция должна быть виртуальной. Операнд приходится добавлять через указатель типа Query*, находящийся в стеке _current_op. Однако способ добавления операнда зависит от типа: для унарных (NotQuery) и бинарных (AndQuery и OrQuery) операций он различен. Наша иерархия классов Query подобные операции не поддерживает.

Оказалось, что анализ предметной области был неполон, в результате чего разработанный интерфейс не согласуется с конкретной реализацией проекта. Нельзя сказать, что анализ был неправильным, он просто неполон. Эта проблема связана с тем, что этапы анализа, проектирования и реализации отделены друг от друга и не допускают обратной связи и пересмотра. Но хотя мы не можем все продумать и все предвидеть, необходимо отличать неизбежные неправильные шаги от ошибок, обусловленных собственной невнимательностью или нехваткой времени.

В таком случае мы должны либо сами модифицировать иерархию классов Query, либо договориться, чтобы это сделали за нас. В данной ситуации мы, как авторы всей системы, сами изменим код, модифицировав конструкторы подтипов и включив виртуальную функцию-член add_op() для добавления операндов после определения оператора (мы покажем, как она применяется, чуть ниже, при рассмотрении функций evalRParen() и evalWord()).

17.7.1. Определение класса UserQuery

Объект класса UserQuery можно инициализировать указателем на вектор строк, представляющий запрос пользователя, или передать ему адрес этого вектора позже, с помощью функции-члена query(). Это позволяет использовать один объект для нескольких запросов. Фактическое построение иерархии классов Query выполняется функцией eval_query():

```
// определить объект, не имея запроса пользователя
UserQuery user_query;

string text;
vector<string> query_text;

// обработать запросы пользователя
do {
   while( cin >> text )
       query_text.push_back( text );

   // передать запрос объекту UserQuery
   user_query.query( &query_text );

   // вычислить результат запроса и вернуть
   // корень иерархии Query*
   Query *query = user_query.eval_query();
}

while ( /* пользователь продолжает формулировать запросы */ );
```

Вот определение нашего класса UserQuery:

```
#ifndef USER_QUERY_H
#define USER_QUERY_H

#include <string>
#include <vector>
#include <map>
#include <stack>

typedef pair<short,short>           location;
typedef vector<location,allocator>  loc;

#include "Query.h"

class UserQuery {
public:
          UserQuery( vector< string,allocator > *pquery = 0 )
          :  _query( pquery ), _eval( 0 ), _paren( 0 ) {}

          Query *eval_query();      // строит иерархию
          void   query( vector< string,allocator > *pq );
          void   displayQuery();

          static void word_map( map<string,loc*,less<string>,allocator>
          *pwm ) {
          if ( !_word_map ) _word_map = pwm;
          }

private:
          enum QueryType { WORD = 1, AND, OR, NOT, RPAREN, LPAREN };

          QueryType evalQueryString( const string &query );
          void      evalWord( const string &query );
          void      evalAnd();
          void      evalOr();
          void      evalNot();
          void      evalRParen();
          bool      integrity_check();

          int             _paren;
          Query           *_eval;
          vector<string> *_query;

          stack<Query*, vector<Query*> > _query_stack;
          stack<Query*, vector<Query*> > _current_op;
     static short _lparenOn, _rparenOn;
     static map<string,loc*,less<string>,allocator> *_word_map;
};

#endif
```

Обратите внимание, что два объявленных нами стека содержат указатели на объекты типа `Query`, а не сами объекты. Хотя правильное поведение обеспечивается обеими реализациями, хранение объектов значительно менее эффективно, поскольку каждый объект (и его операнды) должен быть почленно скопирован в стек (напомним, что операнды копируются виртуальной функцией `clone()`) только для того, чтобы вскоре быть уничтоженным. Если мы не собираемся модифицировать объекты, помещаемые в контейнер, то хранение указателей на них намного эффективнее.

Ниже показаны реализации различных встроенных операций `eval`. Операции `evalAnd()` и `evalOr()` выполняют следующие шаги. Сначала объект извлекается из стека

_query_stack (напомним, что для класса stack, определенного в стандартной библиотеке, это требует двух операций: top() для получения элемента и pop() для удаления его из стека). Затем из хипа выделяется память для объекта класса AndQuery или OrQuery, и указатель на него передается объекту, извлеченному из стека. Каждая операция передает объекту AndQuery или OrQuery счетчики левых или правых скобок, необходимые ему для вывода своего содержимого. И наконец неполный оператор

```
inline void
UserQuery::
evalAnd()
{
          Query *pop = _query_stack.top(); _query_stack.pop();
          AndQuery *pq = new AndQuery( pop );

          if ( _lparenOn )
        { pq->lparen( _lparenOn ); _lparenOn = 0; }
          if ( _rparenOn )
        { pq->rparen( _rparenOn ); _rparenOn = 0; }

          _current_op.push( pq );
}

inline void
UserQuery::
evalOr()
{
          Query *pop = _query_stack.top(); _query_stack.pop();
          OrQuery *pq = new OrQuery( pop );

          if ( _lparenOn )
        { pq->lparen( _lparenOn ); _lparenOn = 0; }

          if ( _rparenOn )
        { pq->rparen( _rparenOn ); _rparenOn = 0; }

          _current_op.push( pq );
```

помещается в стек _current_op:

```
}
```

Операция evalNot() работает следующим образом. В хипе создается новый объект класса NotQuery, которому передаются счетчики левых и правых скобок для правильного

```
inline void
UserQuery::
evalNot()
{
          NotQuery *pq = new NotQuery;

          if ( _lparenOn )
        { pq->lparen( _lparenOn ); _lparenOn = 0; }
          if ( _rparenOn )
        { pq->rparen( _rparenOn ); _rparenOn = 0; }

          _current_op.push( pq );
```

отображения содержимого. Затем неполный оператор помещается в стек _current_op:

```
}
```

При обнаружении закрывающей скобки вызывается операция evalRParen(). Если число активных левых скобок больше числа элементов в стеке _current_op, то ничего не происходит. В противном случае выполняются следующие действия. Из стека _query_stack извлекается текущий еще не присоединенный к оператору операнд, а из стека _current_op – текущий неполный оператор. Вызывается виртуальная функция add_op() класса Query, которая их объединяет. И наконец полный оператор помещается

```
inline void
UserQuery::
evalRParen()
{
          if ( _paren < _current_op.size() )
          {
          Query *poperand = _query_stack.top();
           _query_stack.pop();

          Query *pop = _current_op.top();
           _current_op.pop();
          pop->add_op( poperand );
          _query_stack.push( pop );
          }
```

в стек _query_stack:

```
}
```

Операция evalWord() выполняет следующие действия. Она ищет указанное слово в отображении _word_map взятых из файла слов на векторы позиций. Если слово найдено, берется его вектор позиций и в хипе посредством конструктора с двумя параметрами создается новый объект NameQuery. В противном случае объект порождается с помощью конструктора с одним параметром. Если число элементов в стеке _current_op меньше либо равно числу встреченных ранее скобок, то нет неполного оператора, ожидающего операнда типа NameQuery, поэтому новый объект помещается в стек _query_stack. Иначе из стека _current_op извлекается неполный оператор, к которому с помощью виртуальной функции add_op() присоединяется операнд NameQuery, после чего ставший полным оператор помещается в стек _query_stack:

```
inline void
UserQuery::
evalWord( const string &query )
{
          NameQuery *pq;
          loc       *ploc;

          if ( ! _word_map->count( query ))
          pq = new NameQuery( query );
          else {
                ploc = ( *_word_map )[ query ];
          pq = new NameQuery( query, *ploc );
          }

          if ( _current_op.size() <= _paren )
          _query_stack.push( pq );
          else {
          Query *pop = _current_op.top();
           _current_op.pop();
          pop->add_op( pq );
          _query_stack.push( pop );
          }

}
```

Упражнение 17.21

Напишите деструктор, копирующий конструктор и копирующий оператор присваивания для класса `UserQuery`.

Упражнение 17.22

Напишите функции `print()` для класса `UserQuery`. Обоснуйте свой выбор того, что она выводит.

17.8. Соберем все вместе

Функция `main()` для нашего приложения текстового поиска выглядит следующим образом:

```
#include "TextQuery.h"

int main()
{
   TextQuery tq;

   tq.build_up_text();
   tq.query_text();
}
```

Функция-член `build_text_map()` – это не что иное, как переименованная функция `doit()` из раздела 6.14:

```
inline void
TextQuery::
build_text_map()
{
   retrieve_text();
   separate_words();
   filter_text();
   suffix_text();
   strip_caps();
   build_word_map();
}
```

Функция-член `query_text()` заменяет одноименную функцию из раздела 6.14. В первоначальной реализации в ее обязанности входили прием запроса от пользователя и вывод ответа. Мы решили сохранить за `query_text()` эти задачи, но реализовать ее по-другому[19]:

19 Полный текст программы можно найти на FTP-сайте издательства Addison-Wesley по адресу, указанному на задней стороне обложки.

```
void
TextQuery::query_text()
{
    // локальные объекты:
    *
    * text: содержит все слова запроса
    * query_text: вектор для хранения пользовательского запроса
    * caps: фильтр для поддержки преобразования
    * прописных букв в строчные
    *
    * user_query: объект UserQuery, в котором инкапсулировано
    *             собственно вычисление ответа на запрос
    */
        string text;
        string caps( "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );
        vector<string, allocator> query_text;
        UserQuery user_query;

    // инициализировать статические члены UserQuery
        NotQuery::all_locs( text_locations->second );
        AndQuery::max_col( &line_cnt );
        UserQuery::word_map( word_map );

        do {
         // удалить предыдущий запрос, если он был
        query_text.clear();

         cout << "Введите запрос. Пожалуйста, разделяйте все его "
              << "элементы пробелами.\n"
             << "Запрос (или весь сеанс) завершается точкой ( . ).\n\n"
              << "==> ";

        /*
         * прочитать запрос из стандартного ввода,
         * преобразовать все заглавные буквы, после чего
         * упаковать его в query_text ...
         *
         * примечание: здесь производятся все действия по
         * обработке запроса, связанные собственно с текстом ...
         */
              while( cin  >> text )
        {
            if ( text == "." )
                break;

            string::size_type pos = 0;
             while (( pos = text.find_first_of( caps, pos ))
                        != string::npos )
                   text[pos] = tolower( text[pos] );

            query_text.push_back( text );
             }

         // теперь у нас есть внутреннее представление запроса
         // обработаем его ...
              if ( ! query_text.empty() )
        {
              // передать запрос объекту UserQuery
                  user_query.query( &query_text );
              // вычислить ответ на запрос
              // вернуть иерархию Query*
              // подробности см. в разделе 17.7

              // query - это член класса TextQuery типа Query*
                  query = user_query.eval_query();

              // вычислить иерархию Query,
              // реализация описана в разделе 17.7
                  query->eval();

              // вывести ответ с помощью
              // функции-члена класса TextQuery
                  display_solution();

              // вывести на терминал пользователя дополнительную
              // пустую строку
                  cout << endl;
```

```
}
```

Тестируя программу, мы применили ее к нескольким текстам. Первым стал короткий рассказ Германа Мелвилла “Bartleby”. Здесь иллюстрируется составной запрос AndQuery, для которого подходящие слова расположены в соседних строках. (Отметим, что слова, заключенные между символами косой черты, предполагаются набранными курсивом.)

```
Введите запрос. Пожалуйста, разделяйте все его элементы пробелами.
Запрос (или весь сеанс) завершается точкой ( . ).
==> John && Jacob && Astor

         john ( 3 ) lines match
         jacob ( 3 ) lines match
         john && jacob ( 3 ) lines match
         astor ( 3 ) lines match
         john && jacob && astor ( 5 ) lines match

Requested query: john && jacob && astor
( 34 ) All who know me consider me an eminently /safe/ man. The late
John Jacob
( 35 ) Astor, a personage little given to poethic enthusiasm, had no
hesitation in
( 38 ) my profession by the late John Jacob Astor, a name which, I admit
I love to
( 40 ) bullion. I will freely add that I was not insensible to the late
John Jacob
( 41 ) Astor's good opinion.
```

Следующий запрос, в котором тестируются скобки и составные операторы, обращен к тексту новеллы “Heart of Darkness” Джозефа Конрада:

```
==> horror || ( absurd && mystery ) || ( North && Pole )

         horror ( 5 ) lines match
         absurd ( 8 ) lines match
         mystery ( 12 ) lines match
         ( absurd && mystery ) ( 1 ) lines match
         horror || ( absurd && mystery ) ( 6 ) lines match
         north ( 2 ) lines match
         pole ( 7 ) lines match
         ( north && pole ) ( 1 ) lines match
         horror || ( absurd && mystery ) || ( north && pole )
                ( 7 ) lines match

Requested query: horror || ( absurd && mystery ) || ( north && pole )
( 257 ) up I will go there.' The North Pole was one of these
( 952 ) horros. The heavy pole had skinned his poor nose
( 3055 ) some lightless region of subtle horrors, where pure,
( 3673 ) " 'The horror! The horror!'
( 3913 ) the whispered cry, 'The horror! The horror! '
( 3957 ) absurd mysteries not fit for a human being to behold.
( 4088 ) wind. 'The horror! The horror!'
```

Последний запрос был обращен к отрывку из романа Генри Джеймса “Portrait of a Lady”. В нем иллюстрируется составной запрос в применении к большому текстовому файлу:

```
==> clever && trick || devious

         clever ( 46 ) lines match
         trick ( 12 ) lines match
         clever && trick ( 2 ) lines match
         devious ( 1 ) lines match
         clever && trick || devious ( 3 ) lines match

Requested query: clever && trick || devious
( 13914 ) clever trick she had guessed. Isabel, as she herself grew older
```

```
( 13935 ) lost the desire to know this lady's clever trick. If she had
( 14974 ) desultory, so devious, so much the reverse of processional.
There were
```

Упражнение 17.23

Реализованная нами обработка запроса пользователя обладает одним недостатком: она не применяет к каждому слову те же предварительные фильтры, что и программа, строящая вектор позиций (см. разделы 6.9 и 6.10). Например, пользователь, который хочет найти слово "maps", обнаружит, что в нашем представлении текста распознается только "map", поскольку существительные во множественном числе приводятся к форме в единственном числе. Модифицируйте функцию `query_text()` так, чтобы она применяла эквивалентные фильтры к словам запроса.

Упражнение 17.24

Поисковую систему можно было бы усовершенствовать, добавив еще одну разновидность запроса "И", которую мы назовем `InclusiveAndQuery` и будем обозначать символом &. Строка текста удовлетворяет условиям запроса, если в ней находятся оба указанных слова, пусть даже не рядом. Например, строка

```
We were her pride of ten, she named us
```

удовлетворяет запросу:

```
pride & ten
```

но не:

```
pride && ten
```

Поддержите запрос `InclusiveAndQuery`.

Упражнение 17.25

Представленная ниже реализация функции `display_solution()` может выводить только в стандартный вывод. Более правильно было бы позволить пользователю самому задавать поток `ostream`, в который надо направить вывод. Модифицируйте `display_solution()` так, чтобы `ostream` можно было задавать. Какие еще изменения необходимо внести в определение класса `UserQuery`?

```
void TextQuery::
display_solution()
{
          cout << "\n"
               << "Requested query: "
               << *query << "\n\n";

          const set<short,less<short>,allocator> *solution = query-
          >solution();

          if ( ! solution->size() ) {
               cout << "\n\t"
            << "Sorry, no matching lines were found in text.\n"
            << endl;
          }

          set<short>::const_iterator
          it = solution->begin(),
          end_it = solution->end();

          for ( ; it != end_it; ++it ) {
          int line = *it;

          // пронумеруем строки с 1 ...
          cout << "( " << line+1 << " ) "
               << (*lines_of_text)[line] << '\n';
          }
          cout << endl;
}
```

Упражнение 17.26

Нашему классу `TextQuery` не хватает возможности принимать аргументы, заданные пользователем в командной строке.

(a) Предложите синтаксис командной строки для нашей поисковой системы.

(b) Добавьте в класс необходимые данные и функции-члены.

(c) Предложите средства для работы с командной строкой (см. пример в разделе 7.8).

Упражнение 17.27

В качестве темы для рабочего проекта рассмотрите следующие усовершенствования нашей поисковой системы:

(a) Реализуйте поддержку, необходимую для представления запроса `AndQuery` в виде одной строки, например “Motion Picture Screen Cartoonists”.

(b) Реализуйте поддержку для ответа на запрос на основе вхождения слов не в строку, а в предложение.

(c) Реализуйте подсистему хранения истории, с помощью которой пользователь мог бы ссылаться на предыдущий запрос по номеру, возможно, комбинируя его с новым запросом.

(d) Вместо того чтобы показывать счетчик найденных и все найденные строки, реализуйте возможность задать диапазон выводимых строк для промежуточных вычислений и для окончательного ответа:

```
==> John && Jacob && Astor
```

```
(1)    john ( 3 ) lines match
(2)    jacob ( 3 ) lines match
(3)    john && jacob ( 3 ) lines match
(4)    astor ( 3 ) lines match
(5)    john && jacob && astor ( 5 ) lines match

// Новая возможность: пусть пользователь укажет, какой запрос выводить
// пользователь вводит число
==> вывести? 3

// Затем система спрашивает, сколько строк выводить
// при нажатии клавиши Enter выводятся все строки,
// но пользователь может также ввести номер одной строки или диапазон
ð          сколько (Enter выводит все, иначе введите номер строки или диапазон)
    1-3
```

18. Множественное и виртуальное наследование

В большинстве реальных приложений на C++ используется открытое наследование от одного базового класса. Можно предположить, что и в наших программах оно в основном будет применяться именно так. Но иногда одиночного наследования не хватает, потому что с его помощью либо нельзя адекватно смоделировать абстракцию предметной области, либо получающаяся модель чересчур сложна и неинтуитивна. В таких случаях следует предпочесть множественное наследование или его частный случай – виртуальное наследование. Их поддержка, имеющаяся в C++, – основная тема настоящей главы.

18.1. Готовим сцену

Прежде чем детально описывать множественное и виртуальное наследование, покажем, зачем оно нужно. Наш первый пример взят из области трехмерной компьютерной графики. Но сначала познакомимся с предметной областью.

В компьютере сцена представляется *графом сцены*, который содержит информацию о геометрии (трехмерные модели), один или более источников освещения (иначе сцена будет погружена во тьму), камеру (без нее мы не можем смотреть на сцену) и несколько трансформационных узлов, с помощью которых позиционируются элементы.

Процесс применения источников освещения и камеры к геометрической модели для получения двумерного изображения, отображаемого на дисплее, называется *рендерингом*. В алгоритме рендеринга учитываются два основных аспекта: природа источника освещения сцены и свойства материалов поверхностей объектов, такие, как цвет, шероховатость и прозрачность. Ясно, что перышки на белоснежных крыльях феи выглядят совершенно не так, как капающие из ее глаз слезы, хотя те и другие освещены одним и тем же серебристым светом.

Добавление объектов к сцене, их перемещение, игра с источниками освещения и геометрией – работа компьютерного художника. Наша задача – предоставить интерактивную поддержку для манипуляций с графом сцены на экране. Предположим, что в текущей версии своего инструмента мы решили воспользоваться каркасом приложений Open Inventor для C++ (см. [WERNECKE94]), но с помощью подтипизации расширили его, создав собственные абстракции нужных нам классов. Например, Open Inventor располагает тремя встроенными источниками освещения, производными от

```
class SoSpotLight : public SoLight { ... }
class SoPointLight : public SoLight { ... }
```

абстрактного базового класса `SoLight`:

```
class SoDirectionalLight : public SoLight { ... }
```

Префикс `So` служит для того, чтобы дать уникальные имена сущностям, которые в области компьютерной графики весьма распространены (данный каркас приложений

проектировался еще до появления пространств имен). *Точечный источник* (point light) – это источник света, излучающий, как солнце, во всех направлениях. *Направленный источник* (directional light) – источник света, излучающий в одном направлении. *Прожектор* (spotlight) – источник, испускающий узконаправленный конический пучок, как обычный театральный прожектор.

По умолчанию Open Inventor осуществляет рендеринг графа сцены на экране с помощью библиотеки OpenGL (см. [NEIDER93]). Для интерактивного отображения этого достаточно, но почти все изображения, сгенерированные для киноиндустрии, сделаны с помощью средства RenderMan (см. [UPSTILL90]). Чтобы добавить поддержку такого алгоритма рендеринга мы, в частности, должны реализовать собственные специальные

```
class RiSpotLight : public SoSpotLight { ... }
class RiPointLight : public SoPointLight { ... }
```

подтипы источников освещения:

```
class RiDirectionalLight : public SoDirectionalLight { ... }
```

Новые подтипы содержат дополнительную информацию, необходимую для рендеринга с помощью `RenderMan`. При этом базовые классы `Open Inventor` по-прежнему позволяют выполнять рендеринг с помощью OpenGL. Неприятности начинаются, когда возникает необходимость расширить поддержку теней.

В `RenderMan` направленный источник и прожектор поддерживают отбрасывание тени (поэтому мы называем их источниками освещения, дающими тень, – SCLS), а точечный – нет. Общий алгоритм требует, чтобы мы обошли все источники освещения на сцене и составили *карту теней* для каждого включенного SCLS. Проблема в том, что источники освещения хранятся в графе сцены как полиморфные объекты класса `SoLight`. Хотя мы можем инкапсулировать общие данные и необходимые операции в класс SCLS, непонятно, как включить его в существующую иерархию классов `Open Inventor`.

В поддереве с корнем `SoLight` в иерархии Open Inventor нет такого класса, из которого можно было бы произвести с помощью одиночного наследования класс SCLS так, чтобы в дальнейшем уже от него произвести `SdRiSpotLight` и `SdRiDirectionalLight`. Если не пользоваться множественным наследованием, лучшее, что можно сделать, – это сравнить член класса SCLS с каждым возможным типом SCLS-источника и вызвать

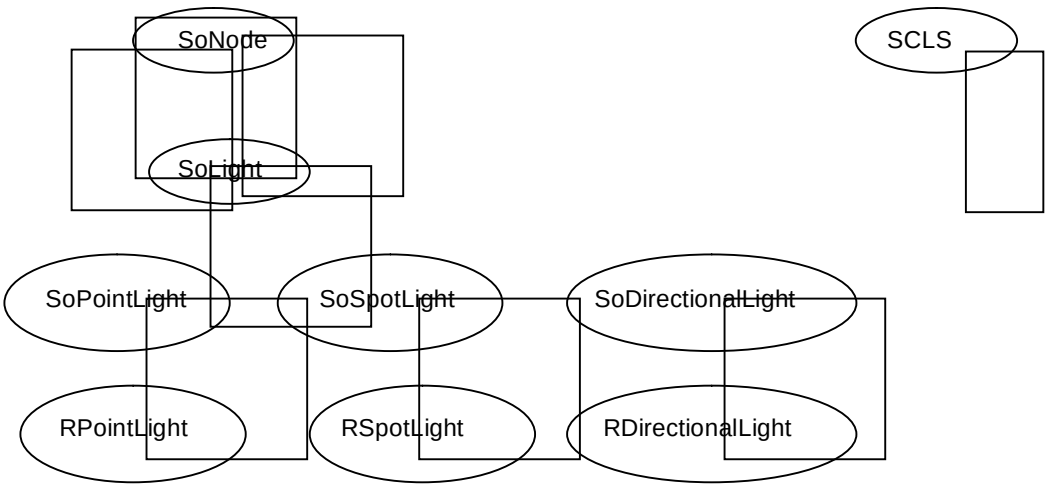
```
SoLight *plight = next_scene_light();

if ( RiDirectionalLight *pdilite =
     dynamic_cast<RiDirectionalLight*>( plight ))
         pdilite->scls.cast_shadow_map();
else
if ( RiSpotLight *pslite =
     dynamic_cast<RiSpotLight*>( plight ))
         pslite->scls.cast_shadow_map();
```

соответствующую операцию:

```
// и так далее
```

(Оператор `dynamic_cast` – это часть механизма идентификации типов во время выполнения (RTTI). Он позволяет опросить тип объекта, адресованного полиморфным указателем или ссылкой. Подробно RTTI будет обсуждаться в главе 19.)

Пользуясь множественным наследованием, мы можем инкапсулировать подтипы SCLS, защитив наш код от изменений при добавлении или удалении источника освещения (см. рис. 18.1).




```
class RiDirectionalLight :
      public SoDirectionalLight, public SCLS { ... };

class RiSpotLight :
      public SoSpotLight, public SCLS { ... };

// ...
SoLight *plight = next_scene_light();
if ( SCLS *pscls = dynamic_cast<SCLS*>(plight))
```

Рис. 18.1. Множественное наследование источников освещения

```
   pscls->cast_shadow_map();
```

Это решение несовершенно. Если бы у нас был доступ к исходным текстам Open Inventor, то можно было бы избежать множественного наследования, добавив к SoLight член-

```
class SoLight : public SoNode {
public:
   void cast_shadow_map()
        { if ( _scls ) _scls->cast_shadow_map(); }
   // ...
protected:
   SCLS *_scls;
};

// ...

SdSoLight *plight = next_scene_light();
```

указатель на SCLS и поддержку операции cast_shadow_map():

```
plight-> cast_shadow_map();
```

Самое распространенное приложение, где используется множественное (и виртуальное) наследование, – это потоковая библиотека ввода/вывода в стандартном C++. Два основных видимых пользователю класса этой библиотеки – `istream` (для ввода) и `ostream` (для вывода). В число их общих атрибутов входят:

- информация о форматировании (представляется ли целое число в десятичной, восьмеричной или шестнадцатеричной системе счисления, число с плавающей точкой – в нотации с фиксированной точкой или в научной нотации и т.д.);
- информация о состоянии (находится ли потоковый объект в нормальном или ошибочном состоянии и т.д.);
- информация о параметрах локализации (отображается ли в начале даты день или месяц и т.д.);
- буфер, где хранятся данные, которые нужно прочитать или записать.

Эти общие атрибуты вынесены в абстрактный базовый класс `ios`, для которого `istream` и `ostream` являются производными.

Класс `iostream` – наш второй пример множественного наследования. Он предоставляет поддержку для чтения и записи в один и тот же файл; его предками являются классы `istream` и `ostream`. К сожалению, по умолчанию он также унаследует два различных экземпляра базового класса `ios`, а нам это не нужно.

Виртуальное наследование решает проблему наследования нескольких экземпляров базового класса, когда нужен только один разделяемый экземпляр. Упрощенная иерархия `iostream` изображена на рис. 18.2.

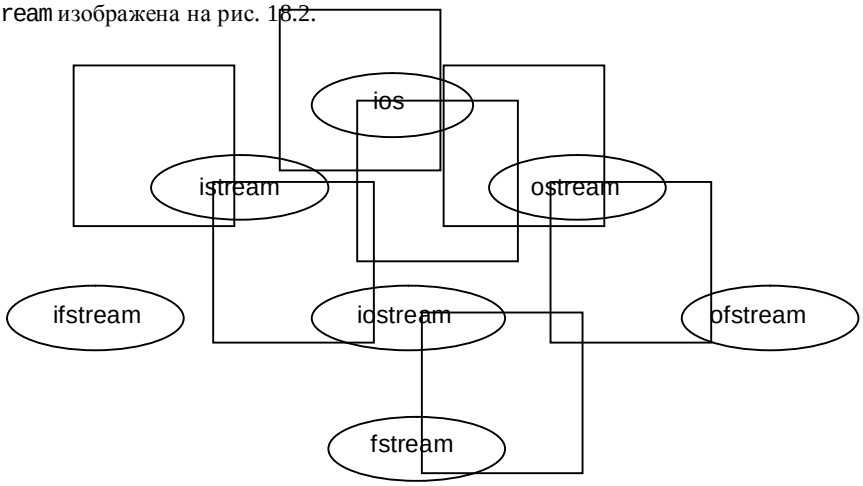


Рис. 18.2. Иерархия виртуального наследования iostream (упрощенная)

Еще один реальный пример виртуального и множественного наследования дают распределенные объектные вычисления. Подробное рассмотрение этой темы см. в серии статей Дугласа Шмидта (Douglas Schmidt) и Стива Виноски (Steve Vinoski) в [LIPPMAN96b].

В данной главе мы рассмотрим использование и поведение механизмов виртуального и множественного наследования. В другой нашей книге, “Inside the C++ Object Model”, описаны более сложные вопросы производительности и дизайна этого аспекта языка.

Для последующего обсуждения мы выбрали иерархию животных в зоопарке. Наши животные существуют на разных уровнях абстракции. Есть, конечно, особи, имеющие

свои имена: Линь-Линь, Маугли или Балу. Каждое животное принадлежит к какому-то виду; скажем, Линь-Линь – это гигантская панда. Виды в свою очередь входят в семейства. Так, гигантская панда – член семейства медведей, хотя, как мы увидим в разделе 18.5, по этому поводу в зоологии долго велись бурные дискуссии. Каждое семейство – член животного мира, в нашем случае ограниченного территорией зоопарка.

На каждом уровне абстракции имеются данные и операции, необходимые для поддержки все более и более широкого круга пользователей. Например, абстрактный класс `ZooAnimal` хранит информацию, общую для всех животных в зоопарке, и предоставляет открытый интерфейс для всех возможных запросов.

Помимо классов, описывающих животных, есть и вспомогательные классы, инкапсулирующие различные абстракции иного рода, например “животные, находящиеся под угрозой вымирания”. Наша реализация класса `Panda` множественно наследует от `Bear` (медведь) и `Endangered` (вымирающие).

18.2. Множественное наследование

Для поддержки множественного наследования синтаксис списка базовых классов

```
class Bear : public ZooAnimal { ... };
```

расширяется: допускается наличие нескольких базовых классов, разделенных запятыми:

```
class Panda : public Bear, public Endangered { ... };
```

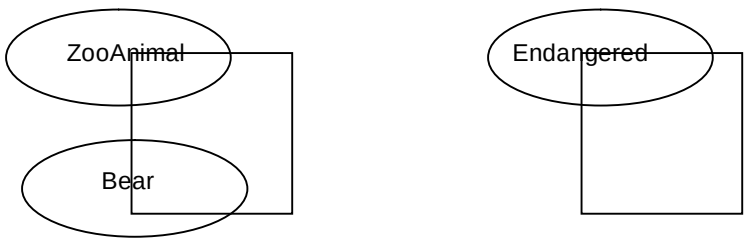
Для каждого из перечисленных базовых классов должен быть указан уровень доступа: `public`, `protected` или `private`. Как и при одиночном наследовании, множественно наследовать можно только классу, определение которого уже встречалось ранее.

Язык не накладывает никаких ограничений на число базовых классов, которым может наследовать производный. На практике чаще всего встречается два класса, один из которых представляет открытый абстрактный интерфейс, а второй – закрытую реализацию (хотя ни один из рассмотренных выше примеров этой модели не следует). Производные классы, наследующие от трех или более базовых, – это пример такого стиля проектирования, когда каждый базовый класс представляет одну *грань* полного интерфейса производного.

В случае множественного наследования объект производного класса содержит по одному подобъекту каждого из своих базовых (см. раздел 17.3). Например, когда мы пишем

```
Panda ying_yang;
```

то объект `ying_yang` будет состоять из подобъекта класса `Bear` (который в свою очередь содержит подобъект `ZooAnimal`), подобъекта `Endangered` и нестатических членов, объявленных в самом классе `Panda`, если таковые есть (см. рис. 18.3).



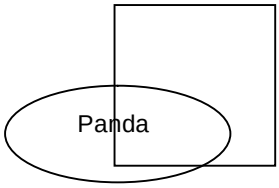


Рис. 18.3. Иерархия множественного наследования класса Panda

Конструкторы базовых классов вызываются в порядке объявления в списке базовых классов. Например, для ying_yang эта последовательность такова: конструктор Bear (но поскольку класс Bear – производный от ZooAnimal, то сначала вызывается конструктор ZooAnimal), затем конструктор Endangered и в самом конце конструктор Panda.

Как отмечалось в разделе 17.4, на порядок вызова *не* влияет ни наличие базовых классов в списке инициализации членов, ни порядок их перечисления. Иными словами, если бы конструктор Bear вызывался неявно и потому не был бы упомянут в списке

```
// конструктор по умолчанию класса Bear вызывается до
// êîíñòðóêòîðà êëàññà Endangered ñ äâóìÿ àðãóìåíòàìè ...

Panda::Panda()
     : Endangered( Endangered::environment,
                   Endangered::critical )
```

инициализации членов, как в следующем примере:

```
{ ... }
```

то все равно конструктор по умолчанию Bear был бы вызван раньше, чем явно заданный в списке конструктор класса Endangered с двумя аргументами.

Порядок вызова деструкторов всегда противоположен порядку вызова конструкторов. В нашем примере деструкторы вызываются в такой последовательности: ~Panda(), ~Endangered(), ~Bear(), ~ZooAnimal().

В разделе 17.3 уже говорилось, что в случае одиночного наследования к открытым и защищенным членам базового класса можно обращаться напрямую (не квалифицируя имя члена именем его класса), как если бы они были членами производного класса. То же самое справедливо и для множественного наследования. Однако при этом можно унаследовать одноименные члены из двух или более базовых классов. В таком случае прямое обращение оказывается неоднозначным и приводит к ошибке компиляции.

Однако такую ошибку вызывает не *потенциальная* неоднозначность неквалифицированного доступа к одному из двух одноименных членов, а лишь попытка фактического обращения к нему (см. раздел 17.4). Например, если в обоих классах Bear и Endangered определена функция-член print(), то инструкция

```
ying_yang.print( cout );
```

приводит к ошибке компиляции, даже если у двух унаследованных функций-членов

```
Error: ying_yang.print( cout ) -- ambiguous, one of
          Bear::print( ostream& )
```

разные списки параметров.

```
Ошибка: ying_yang.print( cout ) -- неоднозначно, одна из
          Bear::print( ostream& )

          Endangered::print( ostream&, int )
          Endangered::print( ostream&, int )
```

Причина в том, что унаследованные функции-члены не образуют множество перегруженных функций внутри производного класса (см. раздел 17.3). Поэтому `print()` разрешается только по имени, а не по типам фактических аргументов. (О том, как производится разрешение, мы поговорим в разделе 18.4.)

В случае одиночного наследования указатель, ссылка или объект производного класса при необходимости автоматически преобразуются в указатель, ссылку или объект базового класса, которому открыто наследует производный. Это остается верным и для множественного наследования. Так, указатель, ссылку или сам объект класса `Panda`

```
extern void display( const Bear& );
extern void highlight( const Endangered& );

Panda ying_yang;

display( ying_yang );   // ïðàâèëüíî
highlight( ying_yang ); // ïðàâèëüíî

extern ostream&
       operator<<( ostream&, const ZooAnimal& );
```

можно преобразовать в указатель, ссылку или объект `ZooAnimal`, `Bear` или `Endangered`:

```
cout << ying_yang << endl;   // правильно
```

Однако вероятность неоднозначных преобразований при множественном наследовании

```
extern void display( const Bear& );
```

намного выше. Рассмотрим, к примеру, две функции:

```
extern void display( const Endangered& );


Panda ying_yang;
```

Неквалифицированный вызов `display()` для объекта класса `Panda`

```
display( ying_yang );   // ошибка: неоднозначность
```

приводит к ошибке компиляции:

```
Error: display( ying_yang ) -- ambiguous, one of
          display( const Bear& );
          display( const Endangered& );
Ошибка: display( ying_yang ) -- неоднозначно, одна из
          display( const Bear& );
          display( const Endangered& );
```

Компилятор не может различить два непосредственных базовых класса с точки зрения преобразования производного. Равным образом применимы обе трансформации. (Мы покажем способ разрешения этого конфликта в разделе 18.4.)

Чтобы понять, какое влияние оказывает множественное наследование на механизм виртуальных функций, определим их набор в каждом из непосредственных базовых классов Panda. (Виртуальные функции введены в разделе 17.2 и подробно обсуждались в

```
class Bear : public ZooAnimal {
public:
   virtual ~Bear();
   virtual ostream& print( ostream& ) const;
   virtual string isA() const;
   // ...
};

class Endangered {
public:
   virtual ~Endangered();
   virtual ostream& print( ostream& ) const;
   virtual void highlight() const;
   // ...
```

разделе 17.5.)

```
};
```

Теперь определим в классе Panda собственный экземпляр print(), собственный

```
class Panda : public Bear, public Endangered
{
public:
   virtual ~Panda();
   virtual ostream& print( ostream& ) const;
   virtual void cuddle();
   // ...
```

деструктор и еще одну виртуальную функцию cuddle():

```
};
```

Множество виртуальных функций, которые можно напрямую вызывать для объекта Panda, представлено в табл. 18.1.

Таблица 18.1. Виртуальные функции для класса Panda

Имя виртуальной функции	Активный экземпляр

	деструктор	Panda::~Panda()
	print(ostream&) const	Panda::print(ostream&)
	isA() const	Bear::isA()
	highlight() const	Endangered::highlight()
	cuddle()	Panda::cuddle()

Когда ссылка или указатель на объект Bear или ZooAnimal инициализируется адресом объекта Panda или ему присваивается такой адрес, то части интерфейса, связанные с

```
Bear *pb = new Panda;

pb->print( cout );      // ïðàâèëüíî: Panda::print(ostream&)
pb->isA();              // ïðàâèëüíî: Bear::isA()
pb->cuddle();           // îøèáêà: ýòî íå ÷àñòü èíòåðôåéñà Bear
pb->highlight();        // îøèáêà: ýòî íå ÷àñòü èíòåðôåéñà Bear
```

классами Panda и Endangered, становятся недоступны:

```
delete pb;              // правильно: Panda::~Panda()
```

(Обратите внимание, что если бы объекту класса Panda был присвоен указатель на ZooAnimal, то все показанные выше вызовы разрешались бы так же.)

Аналогично, если ссылка или указатель на объект Endangered инициализируется адресом объекта Panda или ему присваивается такой адрес, то части интерфейса,

```
Endangered *pe = new Panda;

pe->print( cout );  // правильно: Panda::print(ostream&)

// îøèáêà: ýòî íå ÷àñòü èíòåðôåéñà Endangered
pe->cuddle();

pe->highlight();        // правильно: Endangered::highlight()
```

связанные с классами Panda и Bear, становятся недоступными:

```
delete pe;              // правильно: Panda::~Panda()
```

Обработка виртуального деструктора выполняется правильно независимо от типа указателя, через который мы уничтожаем объект. Например, во всех четырех инструкциях порядок вызова деструкторов один и тот же – обратный порядку вызова конструкторов:

```
// ZooAnimal *pz = new Panda;
delete pz;

// Bear *pb = new Panda;
delete pb;
// Panda *pp = new Panda;
delete pp;

// Endangered *pe = new Panda;
delete pe;
```

Деструктор класса Panda вызывается с помощью механизма виртуализации. После его выполнения по очереди статически вызываются деструкторы Endangered и Bear, а в самом конце – ZooAnimal.

Почленная инициализация и присваивание объекту производного класса, наследующего нескольким базовым, ведут себя точно так же, как и при одиночном наследовании (см.

```
class Panda : public Bear, public Endangered
{ ... };
```

раздел 17.6). Например, для нашего объявления класса Panda

```
Panda yin_yang;
```

в результате почленной инициализации объекта ling_ling

```
Panda ling_ling = yin_yang;
```

вызывается копирующий конструктор класса Bear (но, так как Bear производный от ZooAnimal, сначала выполняется копирующий конструктор класса ZooAnimal), затем – класса Endangered и только потом – класса Panda. Почленное присваивание ведет себя аналогично.

Упражнение 18.1

Какие из следующих объявлений ошибочны? Почему?

```
(b) class DoublyLinkedList:

(a) class CADVehicle : public CAD, Vehicle { ... };
          public List, public List { ... };
```

```
(c) class iostream:
          private istream, private ostream { ... };
```

Упражнение 18.2

```
class A { ... };
class B : public A { ... };
class C : public B { ... };
class X { ... };
class Y { ... };
class Z : public X, public Y { ... };
```

Дана иерархия, в каждом классе которой определен конструктор по умолчанию:

```
class MI : public C, public Z { ... };
```

Каков порядок вызова конструкторов в таком определении:

```
MI mi;
```

Упражнение 18.3

```
class X { ... };
class A { ... };
class B : public A { ... };
class C : private B { ... };
```

Дана иерархия, в каждом классе которой определен конструктор по умолчанию:

```
class D : public X, public C { ... };
```

Какие из следующих преобразований недопустимы:

```
D *pd = new D;

(a) X *px = pd;   (c) B *pb = pd;

(b) A *pa = pd;   (d) C *pc = pd;
```

Упражнение 18.4

Дана иерархия классов, обладающая приведенным ниже набором виртуальных функций:

```
class Base {
public:
   virtual ~Base();
   virtual ostream& print();
   virtual void debug();
   virtual void readOn();
   virtual void writeOn();
   // ...
};

class Derived1 : virtual public Base {
public:
   virtual ~Derived1();
   virtual void writeOn();
   // ...
};

class Derived2 : virtual public Base {
public:
   virtual ~Derived2();
   virtual void readOn();
   // ...
};

class MI : public Derived1, public Derived2 {
public:
   virtual ~MI();
   virtual ostream& print();
   virtual void debug();
   // ...
};
```

```
Base *pb = new MI;

(a) pb->print();   (c) pb->readOn();   (e) pb->log();
(b) pb->debug();   (d) pb->writeOn();  (f) delete pb;
```

Какой экземпляр виртуальной функции вызывается в каждом из следующих случаев:

Упражнение 18.5

На примере иерархии классов из упражнения 18.4 определите, какие виртуальные функции активны при вызове через pd1 и pd2:

```
(b) MI obj;
```

```
(a) Derived1 *pd1 new MI;

    Derived2 d2 = obj;
```

18.3. Открытое, закрытое и защищенное наследование

Открытое наследование называется еще *наследованием типа*. Производный класс в этом случае является подтипом базового; он замещает реализации всех функций-членов, специфичных для типа базового класса, и наследует общие для типа и подтипа функции. Можно сказать, что производный класс служит примером отношения “ЯВЛЯЕТСЯ”, т.е. предоставляет специализацию более общего базового класса. Медведь (Bear) является животным из зоопарка (ZooAnimal); аудиокнига (AudioBook) является предметом, выдаваемым читателям (LibraryLendingMaterial). Мы говорим, что Bear – это подтип ZooAnimal, равно как и Panda. Аналогично AudioBook – подтип LibBook (библиотечная книга), а оба они – подтипы LibraryLendingMaterial. В любом месте программы, где ожидается базовый тип, можно вместо него подставить открыто унаследованный от него подтип, и программа будет продолжать работать правильно (при условии, конечно, что подтип реализован корректно). Во всех приведенных выше примерах демонстрировалось именно наследование типа.

Закрытое наследование называют также *наследованием реализации*. Производный класс напрямую не поддерживает открытый интерфейс базового, но пользуется его реализацией, предоставляя свой собственный открытый интерфейс.

Чтобы показать, какие здесь возникают вопросы, реализуем класс PeekbackStack,

```
bool
PeekbackStack::
```

который поддерживает выборку из стека с помощью метода peekback():

```
peekback( int index, type &value ) { ... }
```

где value содержит элемент в позиции index, если peekback() вернула true. Если же peekback() возвращает false, то заданная аргументом index позиция некорректна и в value помещается элемент из вершины стека.

В реализации PeekbackStack возможны два типа ошибок:

- реализация абстракции PeekbackStack: некорректная реализация поведения класса;
- реализация представления данных: неправильное управление выделением и освобождением памяти, копированием объектов из стека и т.п.

Обычно стек реализуется либо как массив, либо как связанный список элементов (в стандартной библиотеке по умолчанию это делается на базе двусторонней очереди, хотя вместо нее можно использовать вектор, см. главу 6). Хотелось бы иметь гарантированно правильную (или, по крайней мере, хорошо протестированную и поддерживаемую) реализацию массива или списка, чтобы использовать ее в нашем классе PeekbackStack. Если она есть, то можно сосредоточиться на правильности поведения стека.

У нас есть класс IntArray, представленный в разделе 2.3 (мы временно откажемся от применения класса deque из стандартной библиотеки и от поддержки элементов, имеющих отличный от int тип). Вопрос, таким образом, заключается в том, как лучше всего воспользоваться классом IntArray в нашей реализации PeekbackStack. Можно задействовать механизм наследования. (Отметим, что для этого нам придется модифицировать IntArray, сделав его члены защищенными, а не закрытыми.) Реализация выглядела бы так:

```
#include "IntArray.h"

class PeekbackStack : public IntArray {
private:
          const int static bos = -1;

public:
          explicit PeekbackStack( int size )
                 : IntArray( size ), _top( bos ) {}

          bool empty() const { return _top == bos; }
          bool full()  const { return _top == size()-1; }
          int  top()   const { return _top; }

          int pop() {
          if ( empty() )
              /* îáðàáîòàòü îøèáêó */ ;
          return _ia[ _top-- ];
          }

          void push( int value ) {
          if ( full() )
              /* îáðàáîòàòü îøèáêó */ ;
          _ia[ ++_top ] = value;
          }
          bool peekback( int index, int &value ) const;

private:
           int _top;
};

inline bool
PeekbackStack::
peekback( int index, int &value ) const
{
          if ( empty() )
        /* îáðàáîòàòü îøèáêó */ ;

          if ( index < 0 || index > _top )
     {
          value = _ia[ _top ];
          return false;
          }

          value = _ia[ index ];
          return true;

}
```

К сожалению, программа, которая работает с нашим новым классом PeekbackStack,

```
extern void swap( IntArray&, int, int );
PeekbackStack is( 1024 );

// íåïðàâèëüíîå îòêàç ñîîáùåíèå PeekbackStack
swap(is, i, j);
is.sort();
```

может неправильно использовать открытый интерфейс базового IntArray:

```
is[0] = is[512];
```

Абстракция PeekbackStack должна обеспечить доступ к элементам стека по принципу “последним пришел, первым ушел”. Однако наличие дополнительного интерфейса IntArray не позволяет гарантировать такое поведение.

Проблема в том, что открытое наследование описывается как отношение “ЯВЛЯЕТСЯ”. Но PeekbackStack не является разновидностью массива IntArray, а лишь включает его как часть своей реализации. Открытый интерфейс IntArray не должен входить в открытый интерфейс PeekbackStack.

Закрытое наследование от базового класса представляет собой вид наследования, который нельзя описать в терминах подтипов. В производном классе открытый интерфейс базового становится закрытым. Все показанные выше примеры использования объекта PeekbackStack становятся допустимыми только внутри функций-членов и друзей производного класса.

В приведенном ранее определении PeekbackStack достаточно заменить слово public в списке базовых классов на private. Внутри же самого определения класса public и private следует оставить на своих местах:

```
class PeekbackStack : private IntArray { ... };
```

18.3.1. Наследование и композиция

Реализация класса PeekbackStack с помощью закрытого наследования от IntArray работает, но необходимо ли это? Помогло ли нам наследование в данном случае? Нет.

Открытое наследование – это мощный механизм для поддержки отношения “ЯВЛЯЕТСЯ”. Однако реализация PeekbackStack по отношению к IntArray – пример отношения “СОДЕРЖИТ”. Класс PeekbackStack содержит класс IntArray как часть своей реализации. Отношение “СОДЕРЖИТ”, как правило, лучше поддерживается с помощью *композиции*, а не наследования. Для ее реализации надо один класс сделать членом другого. В нашем случае объект IntArray делается членом PeekbackStack. Вот реализация PeekbackStack на основе композиции:

```
class PeekbackStack {
private:
          const int static bos = -1;

public:
          explicit PeekbackStack( int size ) :
          stack( size ), _top( bos ) {}

          bool empty() const { return _top == bos; }
          bool full()  const { return _top == size()-1; }
          int  top()   const { return _top; }

          int pop() {
          if ( empty() )
              /* обработать ошибку */ ;
          return stack[ _top-- ];
          }

          void push( int value ) {
          if ( full() )
              /* обработать ошибку */ ;
          stack[ ++_top ] = value;
          }
          bool peekback( int index, int &value ) const;

private:
           int _top;
      IntArray stack;
};
inline bool
PeekbackStack::
peekback( int index, int &value ) const
{
          if ( empty() )
        /* обработать ошибку */ ;

          if ( index < 0 || index > _top )
     {
           value = stack[ _top ];
           return false;
          }

          value = stack[ index ];
          return true;

}
```

Решая, следует ли использовать при проектировании класса с отношением “СОДЕРЖИТ” композицию или закрытое наследование, можно руководствоваться такими соображениями:

- если мы хотим заместить какие-либо виртуальные функции базового класса, то должны закрыто наследовать ему;
- если мы хотим разрешить нашему классу ссылаться на класс из иерархии типов, то должны использовать композицию по ссылке (мы подробно расскажем о ней в разделе 18.3.4);
- если, как в случае с классом PeekbackStack, мы хотим воспользоваться готовой реализацией, то композиция по значению предпочтительнее наследования. Если

требуется отложенное выделение памяти для объекта, то следует выбрать композицию по ссылке (с помощью указателя).

18.3.2. Открытие отдельных членов

Когда мы применили закрытое наследование класса PeekbackStack от IntArray, то все защищенные и открытые члены IntArray стали закрытыми членами PeekbackStack. Было бы полезно, если бы пользователи PeekbackStack могли узнать размер стека с помощью такой инструкции:

```
is.size();
```

Разработчик способен оградить некоторые члены базового класса от эффектов неоткрытого наследования. Вот как, к примеру, открывается функция-член size() класса

```
class PeekbackStack : private IntArray {
public:
   // сохранить открытый уровень доступа
   using IntArray::size;
   // ...
```

IntArray:

```
};
```

Еще одна причина для открытия отдельных членов заключается в том, что иногда необходимо разрешить доступ к защищенным членам закрыто унаследованного базового класса при последующем наследовании. Предположим, что пользователям нужен подтип стека PeekbackStack, который может динамически расти. Для этого классу, производному от PeekbackStack, понадобится доступ к защищенным элементам ia и

```
template <class Type>
class PeekbackStack : private IntArray {
public:
   using intArray::size;
   // ...

protected:
   using intArray::size;
   using intArray::ia;
   // ...
```

_size класса IntArray:

```
};
```

Производный класс может лишь вернуть унаследованному члену исходный уровень доступа, но не повысить или понизить его по сравнению с указанным в базовом классе.

На практике множественное наследование очень часто применяется для того, чтобы унаследовать открытый интерфейс одного класса и закрытую реализацию другого. Например, в библиотеку классов Booch Components включена следующая реализация растущей очереди Queue (см. также статью Майкла Вило (Michaeel Vilot) и Грейди Буча (Grady Booch) в [LIPPMAN96b]):

```
template < class item, class container >
class Unbounded_Queue:
      private Simple_List< item >,   // ðåàëèçàöèÿ
      public  Queue< item >          // èíòåðôåéñ

{ ... }
```

18.3.3. Защищенное наследование

Третья форма наследования – это *защищенное наследование*. В таком случае все открытые члены базового класса становятся в производном классе защищенными, т.е. доступными из его дальнейших наследников, но не из любого места программы вне иерархии классов. Например, если бы нужно было унаследовать PeekbackStack от

```
// увы: при этом не ïîääåðæèâàåòñÿ дальнейшее наследование
// PeekbackStack: все члены IntArray теперь закрыты
```

Stack, то закрытое наследование

```
class Stack : private IntArray { ... }
```

было бы чересчур ограничительным, поскольку закрытие членов IntArray в классе Stack делает невозможным их последующее наследование. Для того чтобы поддержать наследование вида:

```
class PeekbackStack : public Stack { ... };
```

класс Stack должен наследовать IntArray защищенно:

```
class Stack : protected IntArray { ... };
```

18.3.4. Композиция объектов

Есть две формы композиции объектов:

- *композиция по значению*, когда членом одного класса объявляется сам объект другого класса. Мы показывали это в исправленной реализации PeekbackStack;
- *композиция по ссылке*, когда членом одного класса является указатель или ссылка на объект другого класса.

Композиция по значению обеспечивает автоматическое управление временем жизни объекта и семантику копирования. Кроме того, прямой доступ к объекту оказывается более эффективным. А в каких случаях следует предпочесть композицию по ссылке?

Предположим, что мы решили с помощью композиции представить класс Endangered. Надо ли определить его объект непосредственно внутри ZooAnimal или сослаться на него с помощью указателя или ссылки? Сначала выясним, все ли объекты ZooAnimal обладают этой характеристикой, а если нет, то может ли она изменяться с течением времени (допустимо ли добавлять или удалять эту характеристику).

Если ответ на первый вопрос положительный, то, как правило, лучше применить композицию по значению. (Как правило, но не всегда, поскольку с точки зрения

эффективности включение больших объектов не оптимально, особенно когда они часто копируются. В таких случаях композиция по ссылке позволит обойтись без ненужных копирований, если применять при этом подсчет ссылок и технику, называемую *копированием при записи*. Увеличение эффективности, правда, достигается за счет усложнения управления объектом. Обсуждение этой техники не вошло в наш вводный курс; тем, кому это интересно, рекомендуем прочитать книгу [KOENIG97], главы 6 и 7.)

Если же оказывается, что только некоторые объекты класса `ZooAnimal` обладают указанной характеристикой, то лучшим вариантом будет композиция по ссылке (скажем, в примере с зоопарком не имеет смысла включать в процветающие виды большой объект, описывающий виды вымирающие).

Поскольку объекта `Endangered` может и не существовать, то представлять его надо указателем, а не ссылкой. (Предполагается, что нулевой указатель не адресует объект. Ссылка же всегда должна именовать определенный объект. В разделе 3.6 это различие объяснялось более подробно.)

Если ответ на второй вопрос положительный, то необходимо задать функции, позволяющие вставить и удалить объект `Endangered` во время выполнения.

В нашем примере лишь небольшая часть всего множества животных в зоопарке находится под угрозой вымирания. Кроме того, по крайней мере теоретически, данная характеристика не является постоянной, и, допустим, в один прекрасный день это может

```
class ZooAnimal {
public:
   // ...
   const Endangered* Endangered() const;
   void addEndangered( Endangered* );
   void removeEndangered();
   // ...
protected:
   Endangered *_endangered;
   // ...
```

перестать грозить панде.

```
};
```

Если предполагается, что наше приложение будет работать на разных платформах, то полезно инкапсулировать всю платформенно-зависимую информацию в иерархию абстрактных классов, чтобы запрограммировать платформенно-независимый интерфейс. Например, для вывода объекта `ZooAnimal` на дисплей UNIX-машины и ПК, можно

```
class DisplayManager { ... };
class DisplayUNIX : public DisplayManager { ... };
```

определить иерархию классов `DisplayManager`:

```
class DisplayPC : public DisplayManager { ... };
```

Наш класс `ZooAnimal` не *является* разновидностью класса `DisplayManager`, но *содержит* экземпляр последнего посредством композиции, а не наследования. Возникает вопрос: использовать композицию по значению или по ссылке?

Композиция по значению не может представить объект `DisplayManager`, с помощью которого можно будет адресовать либо объект `DisplayUNIX`, либо объект `DisplayPC`.

Только ссылка или указатель на объект `DisplayManager` позволят нам полиморфно манипулировать его подтипами. Иначе говоря, объектно-ориентированное программирование поддерживается только композицией по ссылке (подробнее см. [LIPPMAN96a].)

Теперь нужно решить, должен ли член класса `ZooAnimal` быть ссылкой или указателем на `DisplayManager`:

- член может быть объявлен ссылкой лишь в том случае, если при создании объекта `ZooAnimal` имеется реальный объект `DisplayManager`, который не будет изменяться по ходу выполнения программы;
- если применяется стратегия *отложенного выделения памяти*, когда память для объекта `DisplayManager` выделяется только при попытке вывести объект на дисплей, то объект следует представить указателем, инициализировав его значением 0;
- если мы хотим переключать режим вывода во время выполнения, то тоже должны представить объект указателем, который инициализирован нулем. Под *переключением* мы понимаем предоставление пользователю возможности выбрать один из подтипов `DisplayManager` в начале или в середине работы программы.

Конечно, маловероятно, что для каждого подобъекта `ZooAnimal` в нашем приложении будет нужен собственный подтип `DisplayManager` для отображения. Скорее всего мы ограничимся статическим членом в классе `ZooAnimal`, указывающим на объект `DisplayManager`.

Упражнение 18.6

Объясните, в каких случаях имеет место наследование типа, а в каких – наследование

```
(a) Queue : List              // очередь : список
(b) EncryptedString : String  // зашифрованная строка : строка
(c) Gif : FileFormat
(d) Circle : Point            // окружность : точка
(e) Dqueue : Queue, List
```

реализации:

```
(f) DrawableGeom : Geom, Canvas // рисуемая фигура : фигура, холст
```

Упражнение 18.7

Замените член `IntArray` в реализации `PeekbackStack` (см. раздел 18.3.1) на класс `deque` из стандартной библиотеки. Напишите небольшую программу для тестирования.

Упражнение 18.8

Сравните композицию по ссылке с композицией по значению, приведите примеры их использования.

18.4. Область видимости класса и наследование

У каждого класса есть собственная область видимости, в которой определены имена членов и вложенные типы (см. разделы 13.9 и 13.10). При наследовании область видимости производного класса вкладывается в область видимости непосредственного базового. Если имя не удается разрешить в области видимости производного класса, то поиск определения продолжается в области видимости базового.

Именно эта иерархическая вложенность областей видимости классов при наследовании и делает возможным обращение к именам членов базового класса так, как если бы они были членами производного. Рассмотрим сначала несколько примеров одиночного наследования, а затем перейдем к множественному. Предположим, есть упрощенное

```
class ZooAnimal {
public:
   ostream &print( ostream& ) const;

   // сделаны открытыми только ради демонстрации разных случаев
   string is_a;
   int    ival;
private:
   double dval;
```

определение класса `ZooAnimal`:

```
};
```

```
class Bear : public ZooAnimal {
public:
   ostream &print( ostream& ) const;

   // сделаны открытыми только ради демонстрации разных случаев
   string name;
   int    ival;
```

и упрощенное определение производного класса `Bear`:

```
};
```

```
Bear bear;
```

Когда мы пишем:

```
bear.is_a;
```

то имя разрешается следующим образом:

- `bear` – это объект класса `Bear`. Сначала поиск имени `is_a` ведется в области видимости `Bear`. Там его нет.
- Поскольку класс `Bear` производный от `ZooAnimal`, то далее поиск `is_a` ведется в области видимости последнего. Обнаруживается, что имя принадлежит его члену. Разрешение закончилось успешно.

Хотя к членам базового класса можно обращаться напрямую, как к членам производного, они сохраняют свою принадлежность к базовому классу. Как правило, не имеет значения, в каком именно классе определено имя. Но это становится важным, если в базовом и производном классах есть одноименные члены. Например, когда мы пишем:

```
bear.ival;
```

ival – это член класса Bear, найденный на первом шаге описанного выше процесса разрешения имени.

Иными словами, член производного класса, имеющий то же имя, что и член базового, маскирует последний. Чтобы обратиться к члену базового класса, необходимо квалифицировать его имя с помощью оператора разрешения области видимости:

```
bear.ZooAnimal::ival;
```

Тем самым мы говорим компилятору, что объявление ival следует искать в области видимости класса ZooAnimal.

Проиллюстрируем использование оператора разрешения области видимости на несколько абсурдном примере (надеемся, вы никогда не напишете чего-либо подобного в реальном

```
int ival;

int Bear::mumble( int ival )
{
   return ival +        // обращение к параметру
        ::ival +        // обращение к глобальному объекту
        ZooAnimal::ival +
        Bear::ival;
```

коде):

```
}
```

Неквалифицированное обращение к ival разрешается в пользу формального параметра. (Если бы переменная ival не была определена внутри mumble(), то имел бы место доступ к члену класса Bear. Если бы ival не была определена и в Bear, то подразумевался бы член ZooAnimal. А если бы ival не было и там, то речь шла бы о глобальном объекте.)

Разрешение имени члена класса всегда предшествует выяснению того, является ли обращение к нему корректным. На первый взгляд, это противоречит интуиции.

```
int dval;
int Bear::mumble( int ival )
{
   // ошибка: разрешается в пользу закрытого члена ZooAnimal::dval
   return ival + dval;
```

Например, изменим реализацию mumble():

```
}
```

Можно возразить, что алгоритм разрешения должен остановиться на первом допустимом в данном контексте имени, а не на первом найденном. Однако в приведенном примере алгоритм разрешения выполняется следующим образом:

(a) Определено ли dval в локальной области видимости функции-члена класса Bear? Нет.

(b) Определено ли dval в области видимости Bear? Нет.

(c) Определено ли dval в области видимости ZooAnimal? Да. Обращение разрешается в пользу этого имени.

После того как имя разрешено, компилятор проверяет, возможен ли доступ к нему. В данном случае нет: dval является закрытым членом, и прямое обращение к нему из mumble() запрещено. Правильное (и, возможно, имевшееся в виду) разрешение требует явного употребления оператора разрешения области видимости:

```
return ival + ::dval;  // правильно
```

Почему же имя члена разрешается перед проверкой уровня доступа? Чтобы предотвратить тонкие изменения семантики программы в связи с совершенно независимым, казалось бы, изменением уровня доступа к члену. Рассмотрим, например,

```
int dval;
int Bear::mumble( int ival )
{
   foo( dval );
   // ...
```

такой вызов:

```
}
```

Если бы функция foo() была перегруженной, то перемещение члена ZooAnimal::dval из закрытой секции в защищенную вполне могло бы изменить всю последовательность вызовов внутри mumble(), а разработчик об этом даже и не подозревал бы.

Если в базовом и производном классах есть функции-члены с одинаковыми именами и сигнатурами, то их поведение такое же, как и поведение данных-членов: член производного класса лексически скрывает в своей области видимости член базового. Для вызова члена базового класса необходимо применить оператор разрешения области

```
ostream& Bear::print( ostream &os) const
{
   // вызывается ZooAnimal::print(os)
   ZooAnimal::print( os );

   os << name;
   return os;
```

видимости:

```
}
```

18.4.1. Область видимости класса при множественном наследовании

Как влияет множественное наследование на алгоритм просмотра области видимости класса? Все непосредственные базовые классы просматриваются одновременно, что может приводить к неоднозначности в случае, когда в нескольких из них есть одноименные члены. Рассмотрим на нескольких примерах, как возникает неоднозначность и какие меры можно предпринять для ее устранения. Предположим, есть следующий набор классов:

```
class Endangered {
public:
   ostream& print( ostream& ) const;
   void highlight();
   // ...
};

class ZooAnimal {
public:
   bool onExhibit() const;
   // ...
private:
   bool highlight( int zoo_location );
   // ...
};

class Bear : public ZooAnimal {
public:
   ostream& print( ostream& ) const;
   void dance( dance_type ) const;
   // ...
};
```

```
class Panda : public Bear, public Endangered {
public:
   void cuddle() const;
   // ...
};
```

`Panda` объявляется производным от двух классов:

Хотя при наследовании функций `print()` и `highlight()` из обоих базовых классов `Bear` и `Endangered` имеется потенциальная неоднозначность, сообщение об ошибке не выдается до момента явно неоднозначного обращения к любой из этих функций.

В то время как неоднозначность двух унаследованных функций `print()` очевидна с первого взгляда, наличие конфликта между членами `highlight()` удивляет (ради этого пример и составлялся): ведь у них разные уровни доступа и разные прототипы. Более того, экземпляр из `Endangered` – это член непосредственного базового класса, а из `ZooAnimal` – член класса, стоящего на две ступеньки выше в иерархии.

Однако все это не имеет значения (впрочем, как мы скоро увидим, может иметь, но в случае виртуального наследования). `Bear` наследует закрытую функцию-член `highlight()` из `ZooAnimal`; лексически она видна, хотя вызывать ее из `Bear` или `Panda` запрещено. Значит, `Panda` наследует два лексически видимых члена с именем `highlight`, поэтому любое неквалифицированное обращение к этому имени приводит к ошибке компиляции.

Поиск имени начинается в ближайшей области видимости, объемлющей его вхождение. Например, в коде

```
int main()
{
   Panda yin_yang;
   yin_yang.dance( Bear::macarena );
}
```

ближайшей будет область видимости класса Panda, к которому принадлежит yin_yang.

```
void Panda::mumble()
{
   dance( Bear::macarena );
   // ...
}
```

Если же мы напишем:

то ближайшей будет локальная область видимости функции-члена mumble(). Если объявление dance в ней имеется, то разрешение имени на этом благополучно завершится. В противном случае поиск будет продолжен в объемлющих областях видимости.

В случае множественного наследования имитируется одновременный просмотр всех поддеревьев наследования – в нашем случае это класс Endangered и поддерево Bear/ZooAnimal. Если объявление обнаружено только в поддереве одного из базовых классов, то разрешение имени заканчивается успешно, как, например, при таком вызове

```
// правильно: Bear::dance()
```

dance():

```
yin_yang.dance( Bear::macarena );
```

Если же объявление найдено в двух или более поддеревьях, то обращение считается неоднозначным и компилятор выдает сообщение об ошибке. Так будет при

```
int main()
{
   // ошибка: неоднозначность: одна из
   //         Bear::print( ostream& ) const
   //         Endangered::print( ostream& ) const
   Panda yin_yang;
   yin_yang.print( cout );
}
```

неквалифицированном обращении к print():

На уровне программы в целом для разрешения неоднозначности достаточно явно квалифицировать имя нужной функции-члена с помощью оператора разрешения области видимости:

```
int main()
{
   // правильно, но не лучшее решение
   Panda yin_yang;
   yin_yang.Bear::print( cout );
}
```

Предложенный способ неэффективен: теперь пользователь вынужден решать, каково правильное поведение класса Panda; однако лучше, если такого рода ответственность примет на себя проектировщик и класс Panda сам устранит все неоднозначности, свойственные его иерархии наследования. Простейший способ добиться этого – задать квалификацию уже в определении экземпляра в производном классе, указав тем самым

```
inline void Panda::highlight() {
   Endangered::highlight();
}

inline ostream&
Panda::print( ostream &os ) const
{
   Bear::print( os );
   Endangered::print( os );
   return os;
}
```

требуемое поведение:

Поскольку успешная компиляция производного класса, наследующего нескольким базовым, не гарантирует отсутствия скрытых неоднозначностей, мы рекомендуем при тестировании вызывать все функции-члены, даже самые тривиальные.

Упражнение 18.9

Дана следующая иерархия классов:

```
class Base1 {
public:
   // ...
protected:
   int    ival;
   double dval;
   char   cval;
   // ...
private:
   int    *id;
   // ...
};

class Base2 {
public:
   // ...
protected:
   float fval;
   // ...
private:
   double dval;
   // ...
};

class Derived : public Base1 {
public:
   // ...
protected:
   string sval;
   double dval;
   // ...
};

class MI : public Derived, public Base2 {
public:
   // ...
protected:
   int             *ival;
   complex<double> cval;
   // ...
};
```

```
int ival;
double dval;

void MI::
foo( double dval )
{
   int id;
   // ...
}
```

и структура функции-члена `MI::foo()`:

(a) Какие члены видны в классе `MI`? Есть ли среди них такие, которые видны в нескольких базовых?

(b) Какие члены видны в `MI::foo()`?

Упражнение 18.10

Пользуясь иерархией классов из упражнения 18.9, укажите, какие из следующих

```
void MI::
bar()
{
   int sval;
   // вопрос упражнения относится к коду, начинающемуся с этого места ...
}

(a) dval = 3.14159; (d) fval = 0;
(b) cval = 'a';     (e) sval = *ival;
```

присваиваний недопустимы внутри функции-члена `MI::bar()`:

```
(c) id = 1;
```

Упражнение 18.11

```
int id;

void MI::
foobar( float cval )
{
   int dval;
   // вопросы упражнения относятся к коду, начинающемуся с этого места ...
```

Даны иерархия классов из упражнения 18.9 и скелет функции-члена `MI::foobar()`:

```
}
```

(a) Присвойте локальной переменной `dval` сумму значений члена `dval` класса `Base1` и члена `dval` класса `Derived`.

(b) Присвойте вещественную часть члена `cval` класса `MI` члену `fval` класса `Base2`.

(c) Присвойте значение члена `cval` класса `Base1` первому символу члена `sval` класса `Derived`.

Упражнение 18.12

Дана следующая иерархия классов, в которых имеются функции-члены `print()`:

```
class Base {
public:
   void print( string ) const;
   // ...
};

class Derived1 : public Base {
public:
   void print( int ) const;
   // ...
};

class Derived2 : public Base {
public:
   void print( double ) const;
   // ...
};

class MI : public Derived1, public Derived2 {
public:
   void print( complex<double> ) const;
   // ...
};
```

```
MI mi;
string dancer( "Nejinsky" );
```

(a) Почему приведенный фрагмент дает ошибку компиляции?

```
mi.print( dancer );
```

(b) Как изменить определение `MI`, чтобы этот фрагмент компилировался и выполнялся правильно?

18.5. Виртуальное наследование A

По умолчанию наследование в C++ является специальной формой композиции по значению. Когда мы пишем:

```
class Bear : public ZooAnimal { ... };
```

каждый объект `Bear` содержит все нестатические данные-члены подобъекта своего базового класса `ZooAnimal`, а также нестатические члены, объявленные в самом `Bear`. Аналогично, если производный класс является базовым для какого-то другого:

```
class PolarBear : public Bear { ... };
```

то каждый объект `PolarBear` содержит все нестатические члены, объявленные в `PolarBear`, `Bear` и `ZooAnimal`.

В случае одиночного наследования эта форма композиции по значению, поддерживаемая механизмом наследования, обеспечивает компактное и эффективное представление объекта. Проблемы возникают только при множественном наследовании, когда некоторый базовый класс неоднократно встречается в иерархии наследования. Самый

известный реальный пример такого рода – это иерархия классов iostream. Взгляните еще раз на рис. 18.2: istream и ostream наследуют одному и тому абстрактному базовому классу ios, а iostream является производным как от istream, так и от

```
class iostream :
```

ostream.

```
    public istream, public ostream { ... };
```

По умолчанию каждый объект iostream содержит два подобъекта ios: из istream и из ostream. Почему это плохо? С точки зрения эффективности хранение двух копий подобъекта ios – пустая трата памяти, поскольку объекту iostream нужен только один экземпляр. Кроме того, конструктор вызывается для каждого подобъекта. Более серьезной проблемой является неоднозначность, к которой приводит наличие двух экземпляров. Например, любое неквалифицированное обращение к члену класса ios дает ошибку компиляции. Какой экземпляр имеется в виду? Что будет, если классы istream и ostream инициализируют свои подобъекты ios по-разному? Можно ли гарантировать, что в классе iostream используется согласованная пара членов ios? Применяемый по умолчанию механизм композиции по значению не дает таких гарантий.

Для решения данной проблемы язык предоставляет альтернативный механизм композиции по ссылке: *виртуальное наследование*. В этом случае наследуется только один разделяемый подобъект базового класса, независимо от того, сколько раз базовый класс встречается в иерархии наследования. Этот разделяемый подобъект называется *виртуальным базовым классом*. С помощью виртуального наследования снимаются проблемы дублирования подобъектов базового класса и неоднозначностей, к которым такое дублирование приводит.

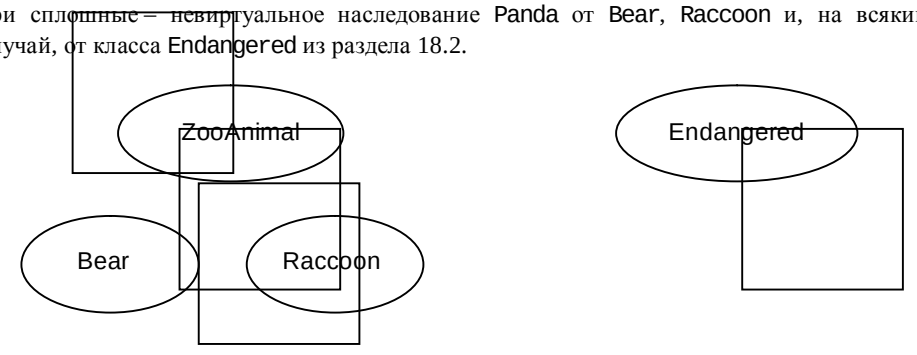
Для изучения синтаксиса и семантики виртуального наследования мы выбрали класс Panda. В зоологических кругах уже на протяжении ста лет периодически вспыхивают ожесточенные споры по поводу того, к какому семейству относить панду: к медведям или к енотам. Поскольку проектирование программного обеспечения призвано обслуживать, в основном, интересы прикладных областей, то самое правильное – произвести класс

```
class Panda : public Bear,
```

Panda от обоих классов:

```
              public Raccoon, public Endangered { ... };
```

Наша виртуальная иерархия наследования Panda показана на рис. 18.4: две пунктирные стрелки обозначают виртуальное наследование классов Bear и Raccoon от ZooAnimal, а три сплошные – невиртуальное наследование Panda от Bear, Raccoon и, на всякий случай, от класса Endangered из раздела 18.2.



¾ ¾ > невиртуальное наследование

- - - -> виртуальное наследование

Рис. 18.4. Иерархия виртуального наследования класса Panda

На данном рисунке показан интуитивно неочевидный аспект виртуального наследования: оно (в нашем случае наследование классов Bear и Raccoon) должно появиться в иерархии раньше, чем в нем возникнет реальная необходимость. Необходимым виртуальное наследование становится только при объявлении класса Panda, но если перед этим базовые классы Bear и Raccoon не наследуют своему базовому виртуально, то проектировщику класса Panda не повезло.

Должны ли мы производить свои базовые классы виртуально просто потому, что где-то ниже в иерархии может потребоваться виртуальное наследование? Нет, это не рекомендуется: снижение производительности и усложнение дальнейшего наследования может оказаться существенным (см. [LIPPMAN96a], где приведены и обсуждаются результаты измерения производительности).

Когда же использовать виртуальное наследование? Чтобы его применение было успешным, иерархия, например библиотека iostream или наше дерево классов Panda, должна проектироваться целиком либо одним человеком, либо коллективом разработчиков.

В общем случае мы не рекомендуем пользоваться виртуальным наследованием, если только оно не решает конкретную проблему проектирования. Однако посмотрим, как все-таки можно его применить.

18.5.1. Объявление виртуального базового класса

Для указания виртуального наследования в объявление базового класса вставляется модификатор virtual. Так, в данном примере ZooAnimal становится виртуальным

```
// взаимное расположение ключевых слов public и virtual
// несущественно

class Bear : public virtual ZooAnimal { ... };
```

базовым для Bear и Raccoon:

```
class Raccoon : virtual public ZooAnimal { ... };
```

Виртуальное наследование не является явной характеристикой самого базового класса, а лишь описывает его отношение к производному. Как мы уже отмечали, виртуальное наследование – это разновидность композиции по ссылке. Иначе говоря, доступ к подобъекту и его нестатическим членам косвенный, что обеспечивает гибкость, необходимую для объединения нескольких виртуально унаследованных подобъектов базовых классов в один разделяемый экземпляр внутри производного. В то же время объектом производного класса можно манипулировать через указатель или ссылку на тип базового, хотя последний является виртуальным. Например, все показанные ниже

преобразования базовых классов Panda выполняются корректно, хотя Panda использует

```
extern void dance( const Bear* );
extern void rummage( const Raccoon* );

extern ostream&
      operator<<( ostream&, const ZooAnimal& );

int main()
{
   Panda yin_yang;

   dance( &yin_yang );   // правильно
   rummage( &yin_yang ); // правильно
   cout << yin_yang;     // правильно
   // ...
```

виртуальное наследование:

```
}
```

Любой класс, который можно задать в качестве базового, разрешается сделать виртуальным, причем он способен содержать все те же элементы, что обычные базовые

```
#include <iostream>
#include <string>

class ZooAnimal;
extern ostream&
      operator<<( ostream&, const ZooAnimal& );
class ZooAnimal {
public:
   ZooAnimal( string name,
              bool onExhibit, string fam_name )
            : _name( name ),
              _onExhibit( onExhibit ), _fam_name( fam_name )
   {}

   virtual ~ZooAnimal();
   virtual ostream& print( ostream& ) const;
   string name() const { return _name; }
   string family_name() const { return _fam_name; }
   // ...

protected:
   bool   _onExhibit;
   string _name;
   string _fam_name;
   // ...
```

классы. Так выглядит объявление ZooAnimal:

```
};
```

К объявлению и реализации непосредственного базового класса при использовании виртуального наследования добавляется ключевое слово virtual. Вот, например, объявление нашего класса Bear:

```
class Bear : public virtual ZooAnimal {
public:
   enum DanceType {
        two_left_feet, macarena, fandango, waltz };

   Bear( string name, bool onExhibit=true )
       : ZooAnimal( name, onExhibit, "Bear" ),
         _dance( two_left_feet )
   {}

   virtual ostream& print( ostream& ) const;
   void dance( DanceType );
   // ...

protected:
   DanceType _dance;
   // ...

};
```

```
class Raccoon : public virtual ZooAnimal {
public:
   Raccoon( string name, bool onExhibit=true )
       : ZooAnimal( name, onExhibit, "Raccoon" ),
         _pettable( false )
   {}

   virtual ostream& print( ostream& ) const;

   bool pettable() const { return _pettable; }
   void pettable( bool petval ) { _pettable = petval; }
   // ...

protected:
   bool _pettable;
   // ...
```

А вот объявление класса Raccoon:

```
};
```

18.5.2. Специальная семантика инициализации

Наследование, в котором присутствует один или несколько виртуальных базовых классов, требует специальной семантики инициализации. Взгляните еще раз на реализации Bear и Raccoon в предыдущем разделе. Видите ли вы, какая проблема связана с порождением класса Panda?

```
class Panda : public Bear,
              public Raccoon, public Endangered {
public:
      Panda( string name, bool onExhibit=true );
      virtual ostream& print( ostream& ) const;

      bool sleeping() const { return _sleeping; }
      void sleeping( bool newval ) { _sleeping = newval; }
      // ...

protected:
      bool _sleeping;
      // ...

};
```

Проблема в том, что конструкторы базовых классов `Bear` и `Raccoon` вызывают конструктор `ZooAnimal` с неявным набором аргументов. Хуже того, в нашем примере значения по умолчанию для аргумента `fam_name` (название семейства) не только отличаются, они еще и неверны для `Panda`.

В случае невиртуального наследования производный класс способен явно инициализировать только свои непосредственные базовые классы (см. раздел 17.4). Так, классу `Panda`, наследующему от `ZooAnimal`, не разрешается напрямую вызвать конструктор `ZooAnimal` в своем списке инициализации членов. Однако при виртуальном наследовании только `Panda` может напрямую вызывать конструктор своего виртуального базового класса `ZooAnimal`.

Ответственность за инициализацию виртуального базового возлагается на *ближайший производный класс*. Например, когда объявляется объект класса `Bear`:

```
Bear winnie( "pooh" );
```

то `Bear` является ближайшим производным классом для объекта `winnie`, поэтому выполняется вызов конструктора `ZooAnimal`, определенный в классе `Bear`. Когда мы пишем:

```
cout << winnie.family_name();
```

будет выведена строка:

```
The family name for pooh is Bear
```

(Название семейства для pooh – это Bear)

Аналогично для объявления

```
Raccoon meeko( "meeko" );
```

`Raccoon` – это ближайший производный класс для объекта `meeko`, поэтому выполняется вызов конструктора `ZooAnimal`, определенный в классе `Raccoon`. Когда мы пишем:

```
cout << meeko.family_name();
```

печатается строка:

```
The family name for meeko is Raccoon
```

(Название семейства для meeko - это Raccoon)

Если же объявить объект типа Panda:

```
Panda yolo( "yolo" );
```

то ближайшим производным классом для объекта yolo будет Panda, поэтому он и отвечает за инициализацию ZooAnimal.

Когда инициализируется объект Panda, то явные вызовы конструктора ZooAnimal в конструкторах классов Raccoon и Bear не выполняются, а вызывается он с теми аргументами, которые указаны в списке инициализации членов объекта Panda. Вот так

```
Panda::Panda( string name, bool onExhibit=true )
          : ZooAnimal( name, onExhibit, "Panda" ),
            Bear( name, onExhibit ),
            Raccoon( name, onExhibit ),
            Endangered( Endangered::environment,
                        Endangered::critical ),
            sleeping( false )
```

выглядит реализация:

```
{}
```

Если в конструкторе Panda аргументы для конструктора ZooAnimal не указаны явно, то вызывается конструктор ZooAnimal по умолчанию либо, если такового нет, выдается ошибка при компиляции определения конструктора Panda.

Когда мы пишем:

```
cout << yolo.family_name();
```

печатается строка:

```
The family name for yolo is Panda
```

(Название семейства для yolo - это Panda)

Внутри определения Panda классы Raccoon и Bear являются промежуточными, а не ближайшими производными. В промежуточном производном классе все прямые вызовы конструкторов виртуальных базовых классов автоматически подавляются. Если бы от Panda был в дальнейшем произведен еще один класс, то сам класс Panda стал бы промежуточным и вызов из него конструктора ZooAnimal также был бы подавлен.

Обратите внимание, что оба аргумента, передаваемые конструкторам Bear и Raccoon, излишни в том случае, когда они выступают в роли промежуточных производных классов. Чтобы избежать передачи ненужных аргументов, мы можем предоставить явный конструктор, вызываемый, когда класс оказывается промежуточным производным. Изменим наш конструктор Bear:

```
class Bear : public virtual ZooAnimal {
public:
   // если выступает в роли ближайшего производного класса
   Bear( string name, bool onExhibit=true )
       : ZooAnimal( name, onExhibit, "Bear" ),
         _dance( two_left_feet )
   {}

   // ... остальное без изменения

protected:
   // если выступает в роли промежуточного производного класса
   Bear() : _dance( two_left_feet ) {}

   // ... остальное без изменения

};
```

Мы сделали этот конструктор защищенным, поскольку он вызывается только из производных классов. Если аналогичный конструктор по умолчанию обеспечен и для

```
Panda::Panda( string name, bool onExhibit=true )
          : ZooAnimal( name, onExhibit, "Panda" ),
            Endangered( Endangered::environment,
                        Endangered::critical ),
            sleeping( false )
```

класса Raccoon, можно следующим образом модифицировать конструктор Panda:

```
{}
```

18.5.3. Порядок вызова конструкторов и деструкторов

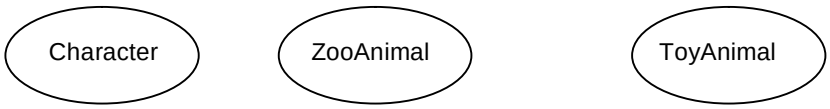
Виртуальные базовые классы всегда конструируются перед невиртуальными, вне зависимости от их расположения в иерархии наследования. Например, в приведенной иерархии у класса TeddyBear (плюшевый мишка) есть два виртуальных базовых: непосредственный – ToyAnimal (игрушечное животное) и экземпляр ZooAnimal, от

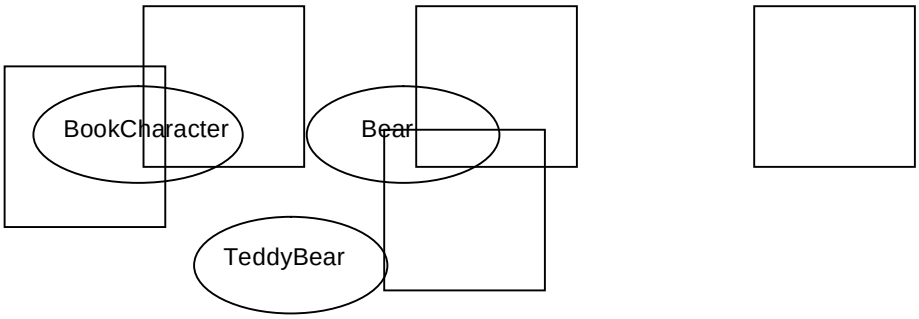
```
class Character { ... };         // персонаж
class BookCharacter : public Character { ... };
                                 // литературный персонаж
class ToyAnimal { ... };         // игрушка

class TeddyBear : public BookCharacter,
                  public Bear, public virtual ToyAnimal
```

которого унаследован класс Bear:

```
                  { ... };
```

Эта иерархия изображена на рис. 18.5, где виртуальное наследование показано пунктирной стрелкой, а невиртуальное – сплошной.





¾ ¾ > невиртуальное наследование

- - - -> виртуальное наследование

Рис. 18.5. Иерархия виртуального наследования класса TeddyBear

Непосредственные базовые классы просматриваются в порядке их объявления при поиске среди них виртуальных. В нашем примере сначала анализируется поддерево наследования `BookCharacter`, затем `Bear` и наконец `ToyAnimal`. Каждое поддерево обходится в глубину, т.е. поиск начинается с корневого класса и продвигается вниз. Так, для поддерева `BookCharacter` сначала просматривается `Character`, а затем `BookCharacter`. Для поддерева `Bear` – `ZooAnimal`, а потом `Bear`.

При описанном алгоритме поиска порядок вызова конструкторов виртуальных базовых классов для `TeddyBear` таков: `ZooAnimal`, потом `ToyAnimal`.

После того как вызваны конструкторы виртуальных базовых классов , настает черед конструкторов невиртуальных, которые вызываются в порядке объявления: `BookCharacter`, затем `Bear`. Перед выполнением конструктора `BookCharacter` вызывается конструктор его базового класса `Character`.

Если имеется объявление:

```
TeddyBear Paddington;
```

```
ZooAnimal();          // виртуальный базовый класс Bear
ToyAnimal();          // непосредственный виртуальный базовый класс
Character();          // невиртуальный базовый класс BookCharacter
BookCharacter();      // непосредственный невиртуальный базовый класс
Bear();               // непосредственный невиртуальный базовый класс
```

то последовательность вызова конструкторов базовых классов будет такой:

```
TeddyBear();          // ближайший производный класс
```

причем за инициализацию `ZooAnimal` и `ToyAnimal` отвечает `TeddyBear` – ближайший производный класс объекта `Paddington`.

Порядок вызова копирующих конструкторов при почленной инициализации (и копирующих операторов присваивания при почленном присваивании) такой же. Гарантируется, что деструкторы вызываются в последовательности, обратной вызову конструкторов.

18.5.4. Видимость членов виртуального базового класса

Изменим наш класс Bear так, чтобы он имел собственную реализацию функции-члена onExhibit(), предоставляемой также ZooAnimal:

```
bool Bear::onExhibit() { ... }
```

Теперь обращение к onExhibit() через объект Bear разрешается в пользу экземпляра,

```
Bear winnie( "любитель меда" );
```

определенного в этом классе:

```
winnie.onExhibit();       // Bear::onExhibit()
```

Обращение же к onExhibit() через объект Raccoon разрешается в пользу функции-

```
Raccoon meeko( "любитель всякой еды" );
```

члена, унаследованной из ZooAnimal:

```
meeko.onExhibit();       // ZooAnimal::onExhibit()
```

Производный класс Panda наследует члены своих базовых классов. Их можно отнести к одной из трех категорий:

- члены виртуального базового класса ZooAnimal, такие, как name() и family(), не замещенные ни в Bear, ни в Raccoon;
- член onExhibit() виртуального базового класса ZooAnimal, наследуемый при обращении через Raccoon и замещенный в классе Bear;
- специализированные в классах Bear и Raccoon экземпляры функции print() из ZooAnimal.

Можно ли, не опасаясь неоднозначности, напрямую обращаться к унаследованным членам из области видимости класса Panda? В случае невиртуального наследования – нет: все неквалифицированные ссылки на имя неоднозначны. Что касается виртуального наследования, то прямое обращение допустимо к любым членам из первой и второй категорий. Например, дан объект класса Panda:

```
Panda spot( "Spottie" );
```

Тогда инструкция

```
spot.name();
```

вызывает разделяемую функцию-член name() виртуального базового ZooAnimal, а инструкция

```
spot.onExhibit();
```

вызывает функцию-член onExhibit() производного класса Bear.

Когда два или более экземпляров члена наследуются разными путями (это относится не только к функциям-членам, но и к данным-членам, а также к вложенным типам) и все они представляют один и тот же член виртуального базового класса, неоднозначности не возникает, поскольку существует единственный разделяемый экземпляр (первая категория). Если один экземпляр представляет член виртуального базового, а другой – член унаследованного от него класса, то неоднозначности также не возникает: специализированному экземпляру из производного класса отдается предпочтение по сравнению с разделяемым экземпляром из виртуального базового (вторая категория). Но если оба экземпляра представляют члены производных классов, то прямое обращение неоднозначно. Лучше всего разрешить эту ситуацию, предоставив замещающий экземпляр в производном классе (третья категория).

Например, при невиртуальном наследовании неквалифицированное обращение к

```
// ошибка: неоднозначно при невиртуальном наследовании
Panda yolo( "любитель бамбука" );
```

onExhibit() через объект Panda неоднозначно:

```
yolo.onExhibit();
```

В данном случае все унаследованные экземпляры имеют равные приоритеты при разрешении имени, поэтому неквалифицированное обращение приводит к ошибке компиляции из-за неоднозначности (см. раздел 18.4.1).

При виртуальном наследовании члену, унаследованному из виртуального базового класса, приписывается меньший приоритет, чем члену с тем же именем, замещенному в производном. Так, унаследованному от Bear экземпляру onExhibit() отдается

```
// правильно: при виртуальном наследовании неоднозначности нет
// вызывается Bear::onExhibit()
```

предпочтение перед экземпляром из ZooAnimal, унаследованному через Raccoon:

```
yolo.onExhibit();
```

Если два или более классов на одном и том же уровне наследования замещают некоторый член виртуального базового, то в производном они будут иметь одинаковый вес. Например, если в Raccoon также определен член onExhibit(), то при обращении к нему из Panda придется квалифицировать имя с помощью оператора разрешения области

```
bool Panda::onExhibit()
{
   return Bear::onExhibit() &&
          Raccoon::onExhibit() &&
          ! _sleeping;
```

видимости:

```
}
```

Упражнение 18.13

Дана иерархия классов:

```
class Class { ... };
class Base : public Class { ... };
class Derived1 : virtual public Base { ... };
class Derived2 : virtual public Base { ... };
class MI : public Derived1,
           public Derived2 { ... };

class Final : public MI, public Class { ... };
```

(a) В каком порядке вызываются конструкторы и деструкторы при определении объекта `Final`?

(b) Сколько подобъектов класса `Base` содержит объект `Final`? А сколько подобъектов `Class`?

```
Base     *pb;
MI       *pmi;
Class    *pc;
Derived2 *pd2;

(i)  pb = new Class;   (iii) pmi = pb;
```

(c) Какие из следующих присваиваний вызывают ошибку компиляции?

```
(ii) pc = new Final;   (iv)  pd2 = pmi;
```

Упражнение 18.14

Дана иерархия классов:

```
class Base {
public:
   bar( int );
   // ...
protected:
   int ival;
   // ...
};

class Derived1 : virtual public Base {
public:
   bar( char );
   foo( char );
   // ...
protected:
   char cval;
   // ...
};

class Derived2 : virtual public Base {
public:
   foo( int );
   // ...
protected:
   int ival;
   char cval;
   // ...
};
```

```
class VMI : public Derived1, public Derived2 {};
```

К каким из унаследованных членов можно обращаться из класса VMI, не квалифицируя имя? А какие требуют квалификации?

Упражнение 18.15

```
class Base {
public:
   Base();
   Base( string );
   Base( const Base& );
   // ...
protected:
   string _name;
```

Дан класс Base с тремя конструкторами:

```
};
```

```
(a) любой из
    class Derived1 : virtual public Vase { ... };
    class Derived2 : virtual public Vase { ... };
(b) class VMI : public Derived1, public Derived2 { ... };
```

Определите соответствующие конструкторы для каждого из следующих классов:

```
(c) class Final : public VMI { ... };
```

18.6. Пример множественного виртуального наследования A

Мы продемонстрируем определение и использование множественного виртуального наследования, реализовав иерархию шаблонов классов Array (см. раздел 2.4) на основе шаблона Array (см. главу 16), модифицированного так, чтобы он стал конкретным базовым классом. Перед тем как приступать к реализации, поговорим о взаимосвязях между шаблонами классов и наследованием.

Конкретизированный экземпляр такого шаблона может выступать в роли явного базового класса:

```
class IntStack : private Array<int> {};
```

```
class Base {};
template <class Type>
```

Разрешается также произвести его от не шаблонного базового класса:

```
    class Derived : public Base {};
```

```
template <class Type>
```

Шаблон может выступать одновременно в роли базового и производного классов:

```
    class Array_RC : public virtual Array<Type> {};
```

В первом примере конкретизированный типом int шаблон Array служит закрытым базовым классом для не шаблонного IntStack. Во втором примере не шаблонный Base служит базовым для любого класса, конкретизированного из шаблона Derived. В третьем примере любой конкретизированный из шаблона Array_RC класс является производным от класса, конкретизированного из шаблона Array. Так, инструкция

```
Array_RC<int> ia;
```

конкретизирует экземпляры шаблонов Array и Array_RC.

```
template < typename Type >
```

Кроме того, сам параметр-шаблон может служить базовым классом [MURRAY93]:

```
     class Persistent : public Type { ... };
```

в данном примере определяется производный устойчивый (persistent) подтип для любого конкретизированного типа. Как отмечает Мюррей (Murray), на Type налагается неявное ограничение: он должен быть типом класса. Например, инструкция

```
Persistent< int > pi;    // ошибка
```

приводит к ошибке компиляции, поскольку встроенный тип не может быть объектом наследования.

Шаблон, выступающий в роли базового класса, должен квалифицироваться полным списком параметров. Если имеется определение:

```
template <class T> class Base {};


template < class Type >
```

то необходимо писать:

```
    class Derived : public Base<Type> {};


// ошибка: Base - это шаблон,
// так что должны быть заданы его аргументы
template < class Type >
```

Такая запись неправильна:

```
    class Derived : public Base {};
```

В следующем разделе шаблон `Array`, определенный в главе 16, выступает в роли виртуального базового класса для подтипа `Array`, контролирующего выход за границы массива; для отсортированного подтипа `Array`; для подтипа `Array`, который обладает обоими указанными свойствами. Однако первоначальное определение шаблона класса `Array` для наследования не подходит:

- все его члены и вспомогательные функции объявлены закрытыми, а не защищенными;
- ни одна из зависящих от типа функций-членов, скажем оператор взятия индекса, не объявлена виртуальной.

Означает ли это, что наша первоначальная реализация была неправильной? Нет. Она была верной на том уровне понимания, которым мы тогда обладали. При реализации шаблона класса `Array` мы еще не осознали необходимость специализированных подтипов. Теперь, однако, определение шаблона придется изменить так (реализации функций-членов при этом останутся теми же):

```
#ifndef ARRAY_H
#define ARRAY_H

#include <iostream>

// необходимо для опережающего объявления operator<<
template <class Type> class Array;

template <class Type> ostream&
          operator<<( ostream &, Array<Type> & );

template <class Type>
class Array {
   static const int ArraySize = 12;
public:
    explicit Array( int sz = ArraySize ) { init( 0, sz ); }
    Array( const Type *ar, int sz )      { init( ar, sz ); }
    Array( const Array &iA )     { init( iA.ia, iA.size()); }
    virtual ~Array()             { delete[] ia; }

    Array& operator=( const Array & );
    int size() const { return _size; }
    virtual void grow();

    virtual void print( ostream& = cout );

    Type at( int ix ) const { return ia[ ix ]; }
    virtual Type& operator[]( int ix ) { return ia[ix]; }

    virtual void sort( int,int );
    virtual int find( Type );
    virtual Type min();
    virtual Type max();

protected:
    void swap( int, int );
    void init( const Type*, int );
    int  _size;
    Type *ia;
};
```

```
#endif
```

Одна из проблем, связанных с таким переходом к полиморфизму, заключается в том, что реализация оператора взятия индекса перестала быть встроенной и сводится теперь к значительно более дорогому вызову виртуальной функции. Так, в следующей функции,

```
int find( const Array< int > &ia, int value )
{
    for ( int ix = 0; ix < ia.size(); ++ix )
          // а теперь вызов виртуальной функции
          if ( ia[ ix ] == value )
                return ix;
    return -1;
```

на какой бы тип она ни ссылалась, было бы достаточно встроенного чтения элемента:

```
}
```

Для повышения производительности мы включили встроенную функцию-член `at()`,обеспечивающую прямое чтение элемента.

18.6.1. Порождение класса, контролирующего выход за границы массива

В функции `try_array()` из раздела 16.13, предназначенной для тестирования нашей

```
int index = iA.find( find_val );
```

предыдущей реализации шаблона класса `Array`, есть две инструкции:

```
Type value = iA[ index ];
```

`find()` возвращает индекс первого вхождения значения `find_val` или `-1`, если значение в массиве не найдено. Этот код некорректен, поскольку в нем не проверяется, что не была возвращена `-1`. Поскольку `-1` находится за границей массива, то каждая инициализация `value` может привести к ошибке. Поэтому мы создадим подтип `Array`, который будет контролировать выход за границы массива, – `Array_RC` и поместим его определение в

```
#ifndef ARRAY_RC_H
#define ARRAY_RC_H

#include "Array.h"

template <class Type>
class Array_RC : public virtual Array<Type> {
public:
    Array_RC( int sz = ArraySize )
            : Array<Type>( sz ) {}
    Array_RC( const Array_RC& r );
    Array_RC( const Type *ar, int sz );
    Type& operator[]( int ix );
};
```

заголовочный файл `Array_RC.h`:

```
#endif
```

Внутри определения производного класса каждая ссылка на спецификатор типа шаблона

```
Array_RC( int sz = ArraySize )
```

базового должна быть квалифицирована списком формальных параметров:

```
        : Array<Type>( sz ) {}
```

```
// ошибка: Array - это не спецификатор типа
```

Такая запись неправильна:

```
Array_RC( int sz = ArraySize ) : Array( sz ) {}
```

Единственное отличие поведения класса Array_RC от базового состоит в том, что оператор взятия индекса контролирует выход за границы массива. Во всех остальных отношениях можно воспользоваться уже имеющейся реализацией шаблона класса Array. Напомним, однако, что конструкторы *не* наследуются, поэтому в Array_RC определен собственный набор из трех конструкторов. Мы сделали класс Array_RC виртуальным наследником класса Array, поскольку предвидели необходимость множественного наследования.

Вот полная реализация функций-членов Array_RC, находящаяся в файле Array_RC.C (определения функций класса Array помещены в заголовочный файл Array.C, поскольку мы пользуемся моделью конкретизации шаблонов с включением, описанной в разделе 16.18):

```
#include "Array_RC.h"
#include "Array.C"
#include <assert.h>

template <class Type>
Array_RC<Type>::Array_RC( const Array_RC<Type> &r )
       :  Array<Type>( r ) {}

template <class Type>
Array_RC<Type>::Array_RC( const Type *ar, int sz )
       :  Array<Type>( ar, sz ) {}

template <class Type>
Type &Array_RC<Type>::operator[]( int ix ) {
        assert( ix >= 0 && ix < Array<Type>::_size );
        return ia[ ix ];
}
```

Мы квалифицировали обращения к членам базового класса Array, например к _size, чтобы предотвратить просмотр Array до момента конкретизации шаблона:

```
Array<Type>::_size;
```

Мы достигаем этого, включая в обращение параметр шаблона. Таким образом, имена в определении Array_RC разрешаются тогда, когда определяется шаблон (за исключением имен, явно зависящих от его параметра). Если встречается неквалифицированное имя _size, то компилятор должен найти его определение, если только это имя не зависит явно от параметра шаблона. Мы сделали имя _size зависящим от параметра шаблона, предварив его именем базового класса Array<Type>. Теперь компилятор не будет пытаться разрешить имя _size до момента конкретизации шаблона. (В определении класса Array_Sort мы приведем другие примеры использования подобных приемов.)

Каждая конкретизация Array_RC порождает экземпляр класса Array. Например:

```
Array_RC<string> sa;
```

конкретизирует параметром string как шаблон Array_RC, так и шаблон Array. Приведенная ниже программа вызывает try_array() (реализацию см. в разделе 16.13), передавая ей объекты подтипа Array_RC. Если все сделано правильно, то выходы за границы массивы будут замечены:

```
#include "Array_RC.C"
#include "try_array.C"

int main()
{
    static int ia[] = { 12,7,14,9,128,17,6,3,27,5 };

    cout << "конкретизация шаблона класса Array_RC<int>\n";
    try_array( iA );

    return 0;
}
```

После компиляции и запуска программа печатает следующее:

```
конкретизация шаблона класса Array_RC<int>

try_array: начальные значения массива
( 10 )< 12, 7, 14, 9, 128, 17
   6, 3, 27, 5 >

try_array: после присваиваний
( 10 )< 128, 7, 14, 9, 128, 128
   6, 3, 27, 3 >

try_array: почленная инициализация
( 10 )< 12, 7, 14, 9, 128, 128
   6, 3, 27, 3 >

try_array: после почленного копирования
( 10 )< 12, 7, 128, 9, 128, 128
   6, 3, 27, 3 >

try_array: после вызова grow
( 10 )< 12, 7, 128, 9, 128, 128
   6, 3, 27, 3, 0, 0
   0, 0, 0, 0 >

искомое значение: 5       возвращенный индекс: -1
Assertion failed: ix >= 0 && ix < _size
```

18.6.2. Порождение класса отсортированного массива

Вторая наша специализация класса `Array` – отсортированный подтип `Array_Sort`. Мы поместим его определение в заголовочный файл `Array_S.h`:

```
#ifndef ARRAY_S_H_
#define ARRAY_S_H_

#include "Array.h"

template <class Type>
class Array_Sort : public virtual Array<Type> {
protected:
    void set_bit()   { dirty_bit = true; }
    void clear_bit() { dirty_bit = false; }

    void check_bit() {
         if ( dirty_bit ) {
              sort( 0, Array<Type>::_size-1 );
              clear_bit();
         }
    }

public:
    Array_Sort( const Array_Sort& );
    Array_Sort( int sz = Array<Type>::ArraySize )
              : Array<Type>( sz )
                { clear_bit();  }

    Array_Sort( const Type* arr, int sz )
              : Array<Type>( arr, sz )
              { sort( 0,Array<Type>::_size-1 ); clear_bit(); }

    Type& operator[]( int ix )
                { set_bit(); return ia[ ix ]; }

    void print( ostream& os = cout ) const
           { check_bit(); Array<Type>::print( os ); }
    Type min() { check_bit(); return ia[ 0 ]; }
    Type max() { check_bit(); return ia[ Array<Type>::_size-1 ]; }

    bool is_dirty() const { return dirty_bit; }
    int  find( Type );
    void grow();

protected:
    bool dirty_bit;
};
```

```
#endif
```

`Array_Sort` включает дополнительный член – `dirty_bit`. Если он установлен в `true`, то не гарантируется, что массив по-прежнему отсортирован. Предоставляется также ряд вспомогательных функций доступа: `is_dirty()` возвращает значение `dirty_bit`; `set_bit()` устанавливает `dirty_bit` в `true`; `clear_bit()` сбрасывает `dirty_bit` в `false`; `check_bit()` пересортировывает массив, если `dirty_bit` равно `true`, после чего сбрасывает его в `false`. Все операции, которые потенциально могут перевести массив в неотсортированное состояние, вызывают `set_bit()`.

При каждом обращении к шаблону `Array` необходимо указывать полный список параметров.

```
Array<Type>::print( os );
```

вызывает функцию-член print() базового класса Array, конкретизированного одновременно с Array_Sort. Например:

```
Array_Sort<string> sas;
```

конкретизирует типом string оба шаблона: Array_Sort и Array.

```
cout << sas;
```

конкретизирует оператор вывода из класса Array, конкретизированного типом string, затем этому оператору передается строка sas. Внутри оператора вывода инструкция

```
ar.print( os );
```

приводит к вызову виртуального экземпляра print() класса Array_Sort, конкретизированного типом string. Сначала вызывается check_bit(), а затем статически вызывается функция-член print() класса Array, конкретизированного тем же типом. (Напомним, что под статическим вызовом понимается разрешение функции на этапе компиляции и – при необходимости – ее подстановка в место вызова.) Виртуальная функция обычно вызывается динамически в зависимости от фактического типа объекта, адресуемого ar. Механизм виртуализации подавляется, если она вызывается явно с помощью оператора разрешения области видимости, как в Array::print(). Это повышает эффективность в случае, когда мы явно вызываем экземпляр виртуальной функции базового класса из экземпляра той же функции в производном, например в print() из класса Array_Sort (см. раздел 17.5).

Функции-члены, определенные вне тела класса, помещены в файл Array_S.C. Объявление может показаться слишком сложным из-за синтаксиса шаблона. Но, если не

```
template <class Type>
Array_Sort<Type>::
Array_Sort( const Array_Sort<Type> &as )
          : Array<Type>( as )
{
    // замечание: as.check_bit() не работает!
    // ---- объяснение см. ниже ...
    if ( as.is_dirty() )
         sort( 0, Array<Type>::_size-1 );
    clear_bit();
}
```

считать списков параметров, оно такое же, как и для обычных классов:

Каждое использование имени шаблона в качестве спецификатора типа должно быть

```
template <class Type>
Array_Sort<Type>::
```

квалифицировано полным списком параметров. Следует писать:

```
Array_Sort( const Array_Sort<Type> &as )
```

а не

```
template <class Type>
Array_Sort<Type>::

Array_Sort<Type>(    // ошибка: это не спецификатор типа
```

поскольку второе вхождение `Array_Sort` синтаксически является именем функции, а не спецификатором типа.

```
if ( as.is_dirty() )
```

Есть две причины, по которым правильна такая запись:

```
    sort( 0, _size );
```

а не просто

```
as.check_bit();
```

Первая причина связана с типизацией: `check_bit()` – это неконстантная функция-член, которая модифицирует объект класса. В качестве аргумента передается ссылка на константный объект. Применение `check_bit()` к аргументу `as` нарушает его константность и потому воспринимается компилятором как ошибка.

Вторая причина: копирующий конструктор рассматривает массив, ассоциированный с `as`, только для того, чтобы выяснить, нуждается ли вновь созданный объект класса `Array_Sort` в сортировке. Напомним, однако, что член `dirty_bit` нового объекта еще не инициализирован. К началу выполнения тела конструктора `Array_Sort` инициализированы только члены `ia` и `_size`, унаследованные от класса `Array`. Этот конструктор должен с помощью `clear_bit()` задать начальные значения дополнительных членов и, вызвав `sort()`, обеспечить специальное поведение подтипа.

```
// альтернативная реализация
template <class Type>
Array_Sort<Type>::
Array_Sort( const Array_Sort<Type> &as )
          : Array<Type>( as )
{
    dirty_bit = as.dirty_bit;
    clear_bit();
```

Конструктор `Array_Sort` можно было бы инициализировать и по-другому:

```
}
```

Ниже приведена реализация функции-члена `grow()`.[1] Наша стратегия состоит в том, чтобы воспользоваться имеющейся в базовом классе `Array` реализацией для выделения дополнительной памяти, а затем пересортировать элементы и сбросить `dirty_bit`:

---

1 Здесь есть потенциальная опасность появления висячей ссылки, если пользователь сохранит адрес какого-либо элемента исходного массива перед тем, как `grow()` скопирует массив в новую область памяти. См. статью Тома Каргилла в [LIPPMAN96b].

```
template <class Type>
void Array_Sort<Type>::grow()
{
    Array<Type>::grow();
    sort( 0, Array<Type>::_size-1 );
    clear_bit();
}
```

```
template <class Type>
int Array_Sort<Type>::find( const Type &val )
{
     int low = 0;
     int high = Array<Type>::_size-1;
     check_bit();
     while ( low <= high ) {
          int mid = ( low + high )/2;

          if ( val == ia[ mid ] )
               return mid;

          if ( val < ia[ mid ] )
               high = mid-1;
          else low = mid+1;
     }
     return -1;
}
```

Так выглядит реализация двоичного поиска в функции-члене `find()` класса `Array_Sort`:

Протестируем нашу реализацию класса `Array_Sort` с помощью функции `try_array()`. Показанная ниже программа тестирует шаблон этого класса для конкретизаций типами `int` и `string`:

```
#include "Array_S.C"
#include "try_array.C"
#include <string>

main()
{
    static int ia[ 10 ] = { 12,7,14,9,128,17,6,3,27,5 };
    static string sa[ 7 ] = {
                "Eeyore", "Pooh", "Tigger",
           "Piglet", "Owl", "Gopher", "Heffalump"
    };

    Array_Sort<int> iA( ia,10 );
    Array_Sort<string> SA( sa,7 );

    cout << "êîíêðåòèçàöèÿ êëàññà Array_Sort<int>"
         << endl;
    try_array( iA );

    cout << "êîíêðåòèçàöèÿ êëàññà Array_Sort<string>"
         << endl;
    try_array( SA );

    return 0;
}
```

При конкретизации типом `string` после компиляции и запуска программа печатает следующий текст (обратите внимание, что попытка вывести элемент с индексом `-1` заканчивается крахом):

```
конкретизация класса Array_Sort<string>

try_array: начальные значения массива
( 7 )< Eeyore, Gopher, Heffalump, Owl, Piglet, Pooh
       Tigger >

try_array: после присваиваний
( 7 )< Eeyore, Gopher, Owl, Piglet, Pooh, Pooh
       Pooh >

try_array: почленная инициализация
( 7 )< Eeyore, Gopher, Owl, Piglet, Pooh, Pooh
       Pooh >

try_array: после почленного копирования
( 7 )< Eeyore, Piglet, Owl, Piglet, Pooh, Pooh
       Pooh >
try_array: после вызова grow
( 7 )< <empty>, <empty>, <empty>, <empty>, Eeyore, Owl
       Piglet, Piglet, Pooh, Pooh, Pooh >

искомое значение: Tigger          возвращенный индекс: -1
Memory fault (coredump)
```

После почленного копирования массив *не* отсортирован, поскольку виртуальная функция вызывалась через объект, а не через указатель или ссылку. Как было сказано в разделе 17.5, в таком случае вызывается экземпляр функции из класса именно этого объекта, а не того подтипа, который может находиться в переменной. Поэтому функция `sort()` никогда не будет вызвана через объект `Array`. (Разумеется, мы реализовали такое поведение только в целях демонстрации.)

18.6.3. Класс массива с множественным наследованием

Определим отсортированный массив с контролем выхода за границы. Для этого можно применить множественное наследование от Array_RC и Array_Sort. Вот как выглядит наша реализация (напомним еще раз, что мы ограничились тремя конструкторами и оператором взятия индекса). Определение находится в заголовочном файле

```
#ifndef ARRAY_RC_S_H
#define ARRAY_RC_S_H

#include "Array_S.C"
#include "Array_RC.C"

template <class Type>
class Array_RC_S : public Array_RC<Type>,
                   public Array_Sort<Type>
{
public:
    Array_RC_S( int sz = Array<Type>::ArraySize )
              : Array<Type>( sz )
              { clear_bit(); }

    Array_RC_S( const Array_RC_S &rca )
                : Array<Type>( rca )
           { sort( 0,Array<Type>::_size-1 ); clear_bit(); }

    Array_RC_S( const Type* arr, int sz )
                : Array<Type>( arr, sz )
                { sort( 0,Array<Type>::_size-1 ); clear_bit(); }

    Type& operator[]( int index )
          {
          set_bit();
                return Array_RC<Type>::operator[]( index );
     }
};
```

Array_RC_S.h:

```
#endif
```

Этот класс наследует две реализации каждой интерфейсной функции Array: из Array_Sort и из виртуального базового класса Array через Array_RC (за исключением оператора взятия индекса, для которого из обоих базовых классов наследуется замещенный экземпляр). При невиртуальном наследовании вызов find() был бы помечен компилятором как неоднозначный, поскольку он не знает, какой из унаследованных экземпляров мы имели в виду. В нашем случае замещенным в Array_Sort экземплярам отдается предпочтение по сравнению с экземплярами, унаследованными из виртуального базового класса через Array_RC (см. раздел 18.5.4). Таким образом, при виртуальном наследовании неквалифицированный вызов find() разрешается в пользу экземпляра, унаследованного из класса Array_Sort.

Оператор взятия индекса переопределен в классах Array_RC и Array_Sort, и обе реализации имеют равный приоритет. Поэтому внутри Array_RC_S неквалифицированное обращение к оператору взятия индекса неоднозначно. Класс Array_RC_S должен предоставить собственную реализацию, иначе пользователи не смогут напрямую применять такой оператор к объектам этого класса. Но какова

семантика его вызова в Array_RC_S? При учете отсортированности массива он должен установить в true унаследованный член dirty_bit. А чтобы учесть наследование от класса с контролем выхода за границы массива – проверить указанный индекс. После этого можно возвращать элемент массива с данным индексом. Последние два шага выполняет унаследованный из Array_RC оператор взятия индекса. При обращении

```
return Array_RC<Type>::operator[]( index );
```

он вызывается явно, и механизм виртуализации не применяется. Поскольку это встроенная функция, то при статическом вызове компилятор подставляет ее код в место вызова.

Теперь протестируем нашу реализацию с помощью функции try_array(), передавая ей

```
#include "Array_RC_S.h"
#include "try_array.C"
#include <string>

int main()
{
    static int ia[ 10 ] = { 12,7,14,9,128,17,6,3,27,5 };
    static string sa[ 7 ] = {
              "Eeyore", "Pooh", "Tigger",
              "Piglet", "Owl", "Gopher", "Heffalump"
    };
    Array_RC_S<int> iA( ia,10 );
    Array_RC_S<string> SA( sa,7 );

    cout << "êîíêðåòèçàöèÿ êëàññà Array_RC_S<int>"
         << endl;
    try_array( iA );

    cout << "êîíêðåòèçàöèÿ êëàññà Array_RC_S<string>"
         << endl;
    try_array( SA );

    return 0;
}
```

по очереди классы, конкретизированные из шаблона Array_RC_S типами int и string:

Вот что печатает программа для класса, конкретизированного типом string (теперь ошибка выхода за границы массива перехватывается):

```
конкретизация класса Array_Sort<string>

try_array: начальные значения массива
( 7 )< Eeyore, Gopher, Heffalump, Owl, Piglet, Pooh
       Tigger >

try_array: после присваиваний
( 7 )< Eeyore, Gopher, Owl, Piglet, Pooh, Pooh
       Pooh >

try_array: почленная инициализация
( 7 )< Eeyore, Gopher, Owl, Piglet, Pooh, Pooh
       Pooh >

try_array: после почленного копирования
( 7 )< Eeyore, Piglet, Owl, Piglet, Pooh, Pooh
       Pooh >
```

```
try_array: после вызова grow
( 7 )< <empty>, <empty>, <empty>, <empty>, Eeyore, Owl
       Piglet, Piglet, Pooh, Pooh, Pooh >

искомое значение: Tigger          возвращенный индекс: -1
Assertion failed: ix >= 0 && ix < size
```

Представленная в этой главе реализация иерархии класса `Array` иллюстрирует применение множественного и виртуального наследования. Детально проектирование класса массива описано в [NACKMAN94]. Однако, как правило, достаточно класса `vector` из стандартной библиотеки.

Упражнение 18.16

Добавьте в `Array` функцию-член `spy()`. Она запоминает операции, примененные к объекту класса: число доступов по индексу; количество вызовов каждого члена; какой элемент искали с помощью `find()` и сколько было успешных поисков. Поясните свои проектные решения. Модифицируйте все подтипы `Array` так, чтобы `spy()` можно было использовать и для них тоже.

Упражнение 18.17

Стандартный библиотечный класс `map` (отображение) называют еще ассоциативным массивом, поскольку он поддерживает индексирование значением ключа. Как вы думаете, является ли ассоциативный массив кандидатом на роль подтипа нашего класса `Array`? Почему?

Упражнение 18.18

Перепишите иерархию `Array`, пользуясь контейнерными классами из стандартной библиотеки и применяя обобщенные алгоритмы.

19

19. Применение наследования в C++

При использовании наследования указатель или ссылка на тип базового класса способен адресовать объект любого производного от него класса. Возможность манипулировать такими указателями или ссылками независимо от фактического типа адресуемого объекта называется *полиморфизмом*. В этой главе мы рассмотрим три функции языка, обеспечивающие специальную поддержку полиморфизма. Сначала мы познакомимся с идентификацией типов во время выполнения (RTTI – Run-time Type Identification), которая позволяет программе узнать истинный производный тип объекта, адресованного ссылкой или указателем на тип базового класса. Затем расскажем о влиянии наследования на обработку исключений: покажем, как можно определять их в виде иерархии классов и как обработчики для типа базового класса могут перехватывать исключения производных типов. В конце главы мы вернемся к правилам разрешения перегрузки функций и посмотрим, как наследование влияет на то, какие преобразования типов можно применять к аргументам функции, и на выбор наилучшей из устоявших.

19.1. Идентификация типов во время выполнения

RTTI позволяет программам, которые манипулируют объектами через указатели или ссылки на базовые классы, получить истинный производный тип адресуемого объекта. Для поддержки RTTI в языке C++ есть два оператора:

- оператор `dynamic_cast` поддерживает преобразования типов во время выполнения, обеспечивая безопасную навигацию по иерархии классов. Он позволяет трансформировать указатель на базовый класс в указатель на производный от него, а также преобразовать l-значение, ссылающееся на базовый класс, в ссылку на производный, но только в том случае, если это завершится успешно;
- оператор `typeid` позволяет получить фактический производный тип объекта, адресованного указателем или ссылкой.

Однако для получения информации о типе производного класса операнд любого из операторов `dynamic_cast` или `typeid` должен иметь тип класса, в котором есть хотя бы одна виртуальная функция. Таким образом, операторы RTTI – это события времени выполнения для классов с виртуальными функциями и события времени компиляции для всех остальных типов. В данном разделе мы более подробно познакомимся с их возможностями.

Использование RTTI оказывается необходимым при реализации таких приложений, как отладчики или объектные базы данных, когда тип объектов, которыми манипулирует программа, становится известен только во время выполнения путем исследования RTTI-информации, хранящейся вместе с типами объектов. Однако лучше пользоваться статической системой типов C++, поскольку она безопаснее и эффективнее.

19.1.1. Оператор dynamic_cast

Оператор dynamic_cast можно применять для преобразования указателя, ссылающегося на объект типа класса в указатель на тип класса из той же иерархии. Его также используют для трансформации l-значения объекта типа класса в ссылку на тип класса из той же иерархии. Приведение типов с помощью оператора dynamic_cast, в отличие от других имеющихся в C++ способов, осуществляется во время выполнения программы. Если указатель или l-значение не могут быть преобразованы в целевой тип, то dynamic_cast завершается неудачно. В случае приведения типа указателя признаком неудачи служит возврат нулевого значения. Если же l-значение нельзя трансформировать в ссылочный тип, возбуждается исключение. Ниже мы приведем примеры неудачного выполнения этого оператора.

Прежде чем перейти к более детальному рассмотрению dynamic_cast, посмотрим, зачем его нужно применять. Предположим, что в программе используется библиотека классов для представления различных категорий служащих компании. Входящие в иерархию

```
class employee {
public:
   virtual int salary();
};

class manager : public employee {
public:
   int salary();
};

class programmer : public employee {
public:
   int salary();
};

void company::payroll( employee *pe ) {
   // используется pe->salary()
}
```

классы поддерживают функции-члены для вычисления зарплаты:

В компании есть разные категории служащих. Параметром функции-члена payroll() класса company является указатель на объект employee, который может адресовать один из типов manager или programmer. Поскольку payroll() обращается к виртуальной функции-члену salary(), то вызывается подходящая замещающая функция, определенная в классе manager или programmer, в зависимости от того, какой объект адресован указателем.

Допустим, класс employee перестал удовлетворять нашим потребностям, и мы хотим его модифицировать, добавив еще одну функцию-член bonus(), используемую совместно с salary() при расчете платежной ведомости. Для этого нужно включить новую функцию-член в классы, составляющие иерархию employee:

```
class employee {
public:
   virtual int salary();       // çàðïëàòà
   virtual int bonus();        // ïðåìèÿ
};

class manager : public employee {
public:
   int salary();
};

class programmer : public employee {
public:
   int salary();
   int bonus();
};

void company::payroll( employee *pe ) {
   // èñïîëüçóåòñÿ pe->salary() è pe->bonus()
}
```

Если параметр pe функции payroll() указывает на объект типа manager, то вызывается виртуальная функция-член bonus() из базового класса employee, поскольку в классе manager она не замещена. Если же pe указывает на объект типа programmer, то вызывается виртуальная функция-член bonus() из класса programmer.

После добавления новых виртуальных функций в иерархию классов придется перекомпилировать все функции-члены. Добавить bonus() можно, если у нас есть доступ к исходным текстам функций-членов в классах employee, manager и programmer. Однако если иерархия была получена от независимого поставщика, то не исключено, что в нашем распоряжении имеются только заголовочные файлы, описывающие интерфейс библиотечных классов и объектные файлы с их реализацией, а исходные тексты функций-членов недоступны. В таком случае перекомпиляция всей иерархии невозможна.

Если мы хотим расширить функциональность библиотеки классов, не добавляя новые виртуальные функции-члены, можно воспользоваться оператором dynamic_cast.

Этот оператор применяется для получения указателя на производный класс, чтобы иметь возможность работать с теми его элементами, которые по-другому не доступны. Предположим, что мы расширяем библиотеку за счет добавления новой функции-члена bonus() в класс programmer. Ее объявление можно включить в определение programmer, находящееся в заголовочном файле, а саму функцию определить в одном из своих исходных файлов:

```
class employee {
public:
   virtual int salary();
};

class manager : public employee {
public:
   int salary();
};

class programmer : public employee {
public:
   int salary();
   int bonus();
};
```

Напомним, что payroll() принимает в качестве параметра указатель на базовый класс employee. Мы можем применить оператор dynamic_cast для получения указателя на

```
void company::payroll( employee *pe )
{
   programmer *pm = dynamic_cast< programmer* >( pe );

   // åñëè pe óêàçûâàåò íà îáúåêò òèïà programmer,
   // òî dynamic_cast âûïîëíèòñÿ óñïåøíî è pm áóäåò
   // óêàçûâàòü íà íà÷àëî îáúåêòà programmer
   if ( pm ) {
      // èñïîëüçîâàòü pm äëÿ âûçîâà programmer::bonus()
   }
   // åñëè pe íå óêàçûâàåò íà îáúåêò òèïà programmer,
   // òî dynamic_cast âûïîëíèòñÿ íåóäà÷íî
   // è pm áóäåò ñîäåðæàòü 0
   else {
      // èñïîëüçîâàòü ôóíêöèè-÷ëåíû êëàññà employee
   }
}
```

производный programmer и воспользоваться им для вызова функции-члена bonus():

Оператор

```
dynamic_cast< programmer* >( pe )
```

приводит свой операнд pe к типу programmer*. Преобразование будет успешным, если pe ссылается на объект типа programmer, и неудачным в противном случае: тогда результатом dynamic_cast будет 0.

Таким образом, оператор dynamic_cast осуществляет сразу две операции. Он проверяет, выполнимо ли запрошенное приведение, и если это так, выполняет его. Проверка производится во время работы программы. dynamic_cast безопаснее, чем другие операции приведения типов в C++, поскольку проверяет возможность корректного преобразования.

Если в предыдущем примере pe действительно указывает на объект типа programmer, то операция dynamic_cast завершится успешно и pm будет инициализирован указателем на объект типа programmer. В противном случае pm получит значение 0. Проверив значение

pm, функция company::payroll() может узнать, указывает ли pm на объект programmer. Если это так, то она вызывает функцию-член programmer::bonus() для вычисления премии программисту. Если же dynamic_cast завершается неудачно, то pe указывает на объект типа manager, а значит, необходимо применить более общий алгоритм расчета, не использующий новую функцию-член programmer::bonus().

Оператор dynamic_cast употребляется для безопасного приведения указателя на базовый класс к указателю на производный. Такую операцию часто называют *понижающим приведением* (downcasting). Она применяется, когда необходимо воспользоваться особенностями производного класса, отсутствующими в базовом. Манипулирование объектами производного класса с помощью указателей на базовый обычно происходит автоматически, с помощью виртуальных функций. Однако иногда использовать виртуальные функции невозможно. В таких ситуациях dynamic_cast предлагает альтернативное решение, хотя этот механизм в большей степени подвержен ошибкам, чем виртуализация, и должен применяться с осторожностью.

Одна из возможных ошибок – это работа с результатом dynamic_cast без предварительной проверки на 0: нулевой указатель нельзя использовать для адресации

```
void company::payroll( employee *pe )
{
   programmer *pm = dynamic_cast< programmer* >( pe );

   // ïîòåíöèàëüíàÿ îøèáêà: pm èñïîëüçóåòñÿ áåç ïðîâåðêè çíà÷åíèÿ
   static int variablePay = 0;
   variablePay += pm->bonus();
   // ...
}
```

объекта класса. Например:

Результат, возвращенный dynamic_cast, всегда следует проверять, прежде чем использовать в качестве указателя. Более правильное определение функции

```
void company::payroll( employee *pe )
{
   // âûïîëíèòü dynamic_cast è ïðîâåðèòü ðåçóëüòàò
   if ( programmer *pm = dynamic_cast< programmer* >( pe ) ) {
      // èñïîëüçîâàòü pm äëÿ âûçîâà programmer::bonus()
   }
   else {
      // èñïîëüçîâàòü ôóíêöèè-÷ëåíû êëàññà employee
   }
}
```

company::payroll() могло бы выглядеть так:

Результат операции dynamic_cast используется для инициализации переменной pm внутри условного выражения в инструкции if. Это возможно, так как объявления в условиях возвращают значения. Ветвь, соответствующая истинности условия, выполняется, если pm не равно нулю: мы знаем, что операция dynamic_cast завершилась успешно и pe указывает на объект programmer. В противном случае результатом объявления будет 0 и выполняется ветвь else. Поскольку теперь оператор и проверка его результата находятся в одной инструкции программы, то невозможно случайно вставить

какой-либо код между выполнением dynamic_cast и проверкой, так что pm будет использоваться только тогда, когда содержит правильный указатель.

В предыдущем примере операция dynamic_cast преобразует указатель на базовый класс в указатель на производный. Ее также можно применять для трансформации l-значения типа базового класса в ссылку на тип производного. Синтаксис такого использования dynamic_cast следующий:

```
dynamic_cast< Type & >( lval )
```

где Type& – это целевой тип преобразования, а lval – l-значение типа базового класса. Операнд lval успешно приводится к типу Type& только в том случае, когда lval действительно относится к объекту класса, для которого один из производных имеет тип Type.

Поскольку нулевых ссылок не бывает (см. раздел 3.6), то проверить успешность выполнения операции путем сравнения результата (т.е. возвращенной оператором dynamic_cast ссылки) с нулем невозможно. Если вместо указателей используются ссылки, условие

```
if ( programmer *pm = dynamic_cast< programmer* >( pe ) )
```

нельзя переписать в виде

```
if ( programmer &pm = dynamic_cast< programmer& >( pe ) )
```

Для извещения об ошибке в случае приведения к ссылочному типу оператор dynamic_cast возбуждает исключение. Следовательно, предыдущий пример можно

```
#include <typeinfo>
void company::payroll( employee &re )
{
   try {
      programmer &rm = dynamic_cast< programmer & >( re );
      // èñïîëüçîâàòü rm äëÿ âûçîâà programmer::bonus()
   }
   catch ( std::bad_cast ) {
      // èñïîëüçîâàòü ôóíêöèè-÷ëåíû êëàññà employee
   }
```

записать так:

```
}
```

В случае неудачного завершения ссылочного варианта dynamic_cast возбуждается исключение типа bad_cast. Класс bad_cast определен в стандартной библиотеке; для ссылки на него необходимо включить в программу заголовочный файл <typeinfo>. (Исключения из стандартной библиотеки мы будем рассматривать в следующем разделе.)

Когда следует употреблять ссылочный вариант dynamic_cast вместо указательного? Это зависит только от желания программиста. При его использовании игнорировать ошибку приведения типа и работать с результатом без проверки (как в указательном варианте) невозможно; с другой стороны, применение исключений увеличивает накладные расходы во время выполнения программы (см. главу 11).

19.1.2. Оператор typeid

Второй оператор, входящий в состав RTTI, – это typeid, который позволяет выяснить фактический тип выражения. Если оно принадлежит типу класса и этот класс содержит хотя бы одну виртуальную функцию-член, то ответ может и не совпадать с типом самого выражения. Так, если выражение является ссылкой на базовый класс, то typeid

```
#include <typeinfo>

programmer pobj;
employee &re = pobj;

// ñ ôóíêöèåé name() ìû ïîçíàêîìèìñÿ â ïîäðàçäåëå, ïîñâÿùåííîì type_info
// îíà âîçâðàùàåò C-ñòðîêó "programmer"
```

сообщает тип производного класса объекта:

```
coiut << typeid( re ).name() << endl;
```

Операнд re оператора typeid имеет тип employee. Но так как re – это ссылка на тип класса с виртуальными функциями, то typeid говорит, что тип адресуемого объекта – programmer (а не employee, на который ссылается re). Программа, использующая такой оператор, должна включать заголовочный файл <typeinfo>, что мы и сделали в этом примере.

Где применяется typeid? В сложных системах разработки, например при построении отладчиков, а также при использовании устойчивых объектов, извлеченных из базы данных. В таких системах необходимо знать фактический тип объекта, которым программа манипулирует с помощью указателя или ссылки на базовый класс, например для получения списка его свойств во время сеанса работы с отладчиком или для правильного сохранения или извлечения объекта из базы данных. Оператор typeid допустимо использовать с выражениями и именами любых типов. Например, его операндами могут быть выражения встроенных типов и константы. Если операнд не

```
int iobj;

cout << typeid( iobj ).name() << endl;   // ïå÷àòàåòñÿ: int
```

принадлежит к типу класса, то typeid просто возвращает его тип:

```
cout << typeid( 8.16 ).name() << endl;   // печатается: double
```

Если операнд имеет тип класса, в котором нет виртуальных функций, то typeid

```
class Base {  /* нет виртуальных функций */ };
class Derived : public Base { /* íåò âèðòóàëüíûõ ôóíêöèé */ };

Derived dobj;
Base *pb = &dobj;
```

возвращает тип операнда, а не связанного с ним объекта:

```
cout << typeid( *pb ).name() << endl;  // печатается: Base
```

Операнд typeid имеет тип Base, т.е. тип выражения *pb. Поскольку в классе Base нет виртуальных функций, результатом typeid будет Base, хотя объект, на который указывает pb, имеет тип Derived.

```
#include <typeinfo>

employee *pe = new manager;
employee& re = *pe;
if ( typeid( pe ) == typeid( employee* ) )  // èñòèííî
   // ÷òî-òî ñäåëàòü
/*
if ( typeid( pe ) == typeid( manager* ) )   // ëîæíî
if ( typeid( pe ) == typeid( employee ) )   // ëîæíî
if ( typeid( pe ) == typeid( manager )  )   // ëîæíî
```

Результаты, возвращенные оператором typeid, можно сравнивать. Например:

```
*/
```

Условие в инструкции if сравнивает результаты применения typeid к операнду, являющемуся выражением, и к операнду, являющемуся именем типа. Обратите внимание, что сравнение

```
typeid( pe ) == typeid( employee* )
```

```
// вызов виртуальной функции
```

возвращает истину. Это удивит пользователей, привыкших писать:

```
pe->salary();
```

что приводит к вызову виртуальной функции salary() из производного класса manager. Поведение typeid(pe) не подчиняется данному механизму. Это связано с тем, что pe – указатель, а для получения типа производного класса операндом typeid должен быть тип класса с виртуальными функциями. Выражение typeid(pe) возвращает тип pe, т.е. указатель на employee. Это значение совпадает со значением typeid(employee*), тогда как все остальные сравнения дают ложь.

Только при употреблении выражения *pe в качестве операнда typeid результат будет

```
typeid( *pe ) == typeid( manager )    // истинно
```

содержать тип объекта, на который указывает pe:

```
typeid( *pe ) == typeid( employee )   // ложно
```

В этих сравнениях *pe – выражение типа класса, который имеет виртуальные функции, поэтому результатом применения typeid будет тип адресуемого операндом объекта manager.

Такой оператор можно использовать и со ссылками:

```
typeid( re ) == typeid( manager )     // истинно
typeid( re ) == typeid( employee )    // ложно
typeid( &re ) == typeid( employee* )  // истинно

typeid( &re ) == typeid( manager* )   // ложно
```

В первых двух сравнениях операнд `re` имеет тип класса с виртуальными функциями, поэтому результат применения `typeid` содержит тип объекта, на который ссылается `re`. В последних двух сравнениях операнд `&re` имеет тип указателя, следовательно, результатом будет тип самого операнда, т.е. `employee*`.

На самом деле оператор `typeid` возвращает объект класса типа `type_info`, который определен в заголовочном файле `<typeinfo>`. Интерфейс этого класса показывает, что можно делать с результатом, возвращенным `typeid`. (В следующем подразделе мы подробно рассмотрим этот интерфейс.)

19.1.3. Класс type_info

Точное определение класса `type_info` зависит от реализации, но некоторые его характерные черты остаются неизменными в любой программе на C++:

```
class type_info {
   // ïðåäîòâðàùåíèå çàâèñèò îò ðåàëèçàöèè
private:
   type_info( const type_info& );
   type_info& operator= ( const type_info& );
public:
   virtual ~type_info();

   int operator==( const type_info& );
   int operator!=( const type_info& );

   const char * name() const;
};
```

Поскольку копирующие конструктор и оператор присваивания – закрытые члены класса `type_info`, то пользователь не может создать его объекты в своей программе:

```
#include <typeinfo>

type_info t1;  // îøèáêà: íåò êîíñòðóêòîðà ïî óìîë÷àíèþ
               // îøèáêà: êîïèðóþùèé êîíñòðóêòîð çàêðûò

type_info t2 (typeid( unsigned int ) );
```

Единственный способ создать объект класса `type_info` – воспользоваться оператором `typeid`.

В классе определены также операторы сравнения. Они позволяют сравнивать два объекта `type_info`, а следовательно, и результаты, возвращенные двумя операторами `typeid`. (Мы говорили об этом в предыдущем подразделе.)

```
typeid( re )  == typeid( manager )     // èñòèííî
typeid( *pe ) != typeid( employee )    // ложно
```

Функция name() возвращает С-строку с именем типа, представленного объектом

```
#include <typeinfo>
int main() {
   employee *pe = new manager;

   // ïå÷àòàåò: "manager"
   cout << typeid( *pe ).name() << endl;
}
```

type_info. Этой функцией можно пользоваться в программах следующим образом:

Для работы с функцией-членом name() нужно включить заголовочный файл <typeinfo>.

Имя типа – это единственная информация, которая гарантированно возвращается всеми реализациями C++, при этом используется функция-член name() класса type_info. В начале этого раздела упоминалось, что поддержка RTTI зависит от реализации и иногда в классе type_info бывают дополнительные функции-члены. Чтобы узнать, каким образом обеспечивается поддержка RTTI в вашем компиляторе, обратитесь к справочному руководству по нему. Кроме того, можно получить любую информацию, которую компилятор знает о типе, например:

- список функций-членов класса;
- способ размещения объекта в памяти, т.е. взаимное расположение подобъектов базового и производных классов.

Одним из способов расширения поддержки RTTI является включение дополнительной информации в класс, производный от type_info. Поскольку в классе type_info есть виртуальный деструктор, то оператор dynamic_cast позволяет выяснить, имеется ли некоторое конкретное расширение RTTI. Предположим, что некоторый компилятор предоставляет расширенную поддержку RTTI посредством класса extended_type_info, производного от type_info. С помощью оператора dynamic_cast программа может узнать, принадлежит ли объект типа type_info, возвращенный оператором typeid, к типу extended_type_info. Если да, то пользоваться расширенной поддержкой RTTI разрешено.

```
#include <typeinfo>

// Файл typeinfo содержит определение типа extended_type_info

void func( employee* p )
{
   // понижающее приведение типа type_info* к extended_type_info*
   if ( eti *eti_p = dynamic_cast<eti *>( &typeid( *p ) ) )
   {
      // если dynamic_cast завершается успешно,
      // можно пользоваться информацией из extended_type_info через eti_p
   }
   else
   {
      // если dynamic_cast завершается неудачно,
      // можно пользоваться только стандартным type_info
   }
}
```

Если dynamic_cast завершается успешно, то оператор typeid вернет объект класса extended_type_info, т.е. компилятор обеспечивает расширенную поддержку RTTI, чем программа может воспользоваться. В противном случае допустимы только базовые средства RTTI.

Упражнение 19.1

Дана иерархия классов, в которой у каждого класса есть конструктор по умолчанию и

```
class X { ... };
class A { ... };
class B : public A { ... };
class C : public B { ... };
```

виртуальный деструктор:

```
class D : public X, public C { ... };
```

```
(a) D *pd = new D;
```

Какие из данных операторов dynamic_cast завершатся неудачно?

```
(b) A *pa = new C;
    A *pa = dynamic_cast< A* > ( pd );
```

```
(c) B *pb = new B;
    C *pc = dynamic_cast< C* > ( pa );

    D *pd = dynamic_cast< D* > ( pb );
```

```
(d) A *pa = new D;
    X *px = dynamic_cast< X* > ( pa );
```

Упражнение 19.2

Объясните, когда нужно пользоваться оператором dynamic_cast вместо виртуальной функции?

Упражнение 19.3

Пользуясь иерархией классов из упражнения 19.1, перепишите следующий фрагмент так, чтобы в нем использовался ссылочный вариант dynamic_cast для преобразования *pa в

```
if ( D *pd = dynamic_cast< D* >( pa ) ) {
   // использовать члены D
}
else {
   // использовать члены A
```

тип D&:

```
}
```

Упражнение 19.4

Дана иерархия классов, в которой у каждого класса есть конструктор по умолчанию и

```
class X { ... };
class A { ... };
class B : public A { ... };
class C : public B { ... };
```

виртуальный деструктор:

```
class D : public X, public C { ... };
```

```
(a) A *pa = new D;
    cout << typeid( pa ).name() << endl;

(b) X *px = new D;
    cout << typeid( *px ).name() << endl;

(c) C obj;
    A& ra = cobj;
    cout << typeid( &ra ).name() << endl;

(d) X *px = new D;
    A& ra = *px;
```

Какое имя типа будет напечатано в каждом из следующих случаев:

```
    cout << typeid( ra ).name() << endl;
```

19.2. Исключения и наследование

Обработка исключений – это стандартное языковое средство для реакции на аномальное поведение программы во время выполнения. C++ поддерживает единообразный синтаксис и стиль обработки исключений, а также способы тонкой настройки этого механизма в специальных ситуациях. Основы его поддержки в языке C++ описаны в главе 11, где показано, как программа может возбудить исключение, передать управление его обработчику (если таковой существует) и как обработчики исключений ассоциируются с try-блоками.

Возможности механизма обработки исключений становятся больше, если в качестве исключений использовать иерархии классов. В этом разделе мы расскажем, как писать программы, которые умеют возбуждать и обрабатывать исключения, принадлежащие таким иерархиям.

19.2.1. Исключения, определенные как иерархии классов

В главе 11 мы использовали два типа класса для описания исключений, возбуждаемых

```
class popOnEmpty { ... };
```

функциями-членами нашего класса iStack:

```
class pushOnFull { ... };
```

В реальных программах на C++ типы классов, представляющих исключения, чаще всего организуются в группы, или иерархии. Как могла бы выглядеть вся иерархия для этих классов?

Мы можем определить базовый класс Excp, которому наследуют оба наши класса исключений. Он инкапсулирует данные и функции-члены, общие для обоих

```
class Excp { ... };
class popOnEmpty : public Excp { ... };
```

производных:

```
class pushOnFull : public Excp { ... };
```

Одной из операцией, которые предоставляет базовый класс, является вывод сообщения об

```
class Excp {
public:
   // напечатать сообщение об ошибке
   static void print( string msg ) {
      cerr << msg << endl;
   }
```

ошибке. Эта возможность используется обоими классами, стоящими ниже в иерархии:

```
};
```

Иерархию классов исключений разрешается развивать и дальше. От Excp можно произвести другие классы для более точного описания исключений, обнаруживаемых

```
class Excp { ... };

class stackExcp : public Excp { ... };
   class popOnEmpty : public stackExcp { ... };
```

программой:

```
class mathExcp : public Excp ( ... };
   class zeroOp : public mathExcp { ... };

   class pushOnFull : public stackExcp { ... };
   class divideByZero : public mathExcp { ... };
```

Последующие уточнения позволяют более детально идентифицировать аномальные ситуации в работе программы. Дополнительные классы исключений организуются как слои. По мере углубления иерархии каждый новый слой описывает все более специфичные исключения. Например, первый, самый общий слой в приведенной выше иерархии представлен классом Excp. Второй специализирует Excp, выделяя из него два подкласса: stackExcp (для исключений при работе с нашим iStack) и mathExcp (для исключений, возбуждаемых функциями из математической библиотеки). Третий, самый специализированный слой данной иерархии уточняет классы исключений: popOnEmpty и pushOnFull определяют два вида исключений работы со стеком, а ZeroOp и divideByZero – два вида исключений математических операций.

В последующих разделах мы рассмотрим, как возбуждаются и обрабатываются исключения, представленные классами в нашей иерархии.

19.2.2. Возбуждение исключения типа класса

Теперь, познакомившись с классами, посмотрим, что происходит, когда функция-член

```
void iStack::push( int value )
{
   if ( full() )
      // value сохраняется в объекте-исключении
      throw pushOnFull( value );
   // ...
```

push() нашего iStack возбуждает исключение:

```
}
```

Выполнение инструкции throw инициирует несколько последовательных действий:

1. Инструкция throw создает временный объект типа класса pushOnFull, вызывая его конструктор.
2. С помощью копирующего конструктора генерируется объект-исключение типа pushOnFull – копия временного объекта, полученного на шаге 1. Затем он передается обработчику исключения.
3. Временный объект, созданный на шаге 1, уничтожается до начала поиска обработчика.

Зачем нужно генерировать объект-исключение (шаг 2)? Инструкция

```
throw pushOnFull( value );
```

создает временный объект, который уничтожается в конце работы throw. Но исключение должно существовать до тех пор, пока не будет найден его обработчик, а он может находиться намного выше в цепочке вызовов. Поэтому необходимо скопировать временный объект в некоторую область памяти (*объект-исключение*), которая гарантированно существует, пока исключение не будет обработано. Иногда компилятор создает объект-исключение сразу, минуя шаг 1. Однако стандарт этого не требует, да и не всегда такое возможно.

Поскольку объект-исключение создается путем копирования значения, переданного инструкции throw, то возбужденное исключение всегда имеет такой же тип, как и это

```
void iStack::push( int value ) {
   if ( full() ) {
      pushOnFull except( value );
      stackExcp *pse = &except;
      throw *pse;   // объект-исключение имеет тип stackExcp
   }
   // ...
```

значение:

```
}
```

Выражение *pse имеет тип stackExcp. Тип созданного объекта-исключения – stackExcp, хотя pse ссылается на объект с фактическим типом pushOnFull. Фактический тип объекта, на который ссылается throw, при создании объекта-исключения не учитывается. Поэтому исключение не будет перехвачено catch-обработчиком pushOnFull.

Действия, выполняемые инструкцией throw, налагают определенные ограничения на то, какие классы можно использовать для создания объектов-исключений. Оператор throw в функции-члене push() класса iStack вызовет ошибку компиляции, если:

- в классе pushOnFull нет конструктора, принимающего аргумент типа int, или этот конструктор недоступен;
- в классе pushOnFull есть копирующий конструктор или деструктор, но хотя бы один из них недоступен;
- pushOnFull – это абстрактный базовый класс. Напомним, что программа не может создавать объекты абстрактных классов (см. раздел 17.1).

19.2.3. Обработка исключения типа класса

Если исключения организуются в иерархии, то исключение типа некоторого класса может быть перехвачено обработчиком, соответствующим любому его открытому базовому классу. Например, исключение типа pushOnFull перехватывается обработчиками исключений типа stackExcp или Excp.

```
int main() {
   try {
      // ...
   }
   catch ( Excp ) {
      // обрабатывает исключения popOnEmpty и pushOnFull
   }
   catch ( pushOnFull ) {
      // обрабатывает исключение pushOnFull
   }
```

Здесь порядок catch-обработчиков желательно изменить. Напоминаем, что они просматриваются в порядке появления после try-блока. Как только будет найден обработчик, способный обработать данное исключение, поиск прекращается. В примере выше Excp может обработать исключения типа pushOnFull, а это значит, что специализированный обработчик таких исключений задействован не будет. Правильная

```
catch ( pushOnFull ) {
   // обрабатывает исключение pushOnFull
}
catch ( Excp ) {
   // обрабатывает другие исключения
}
```

последовательность такова:

catch-обработчик для производного класса должен идти первым. Тогда catch-обработчик для базового класса получит управление только в том случае, если более специализированного обработчика не нашлось.

Если исключения организованы в иерархии, то пользователи библиотеки классов могут выбрать в своем приложении уровень детализации при работе с исключениями, возбужденными внутри библиотеки. Например, кодируя функцию main(), мы решили, что исключения типа pushOnFull должны обрабатываться несколько иначе, чем прочие, и потому написали для них специализированный catch-обработчик. Что касается

```
catch ( pushOnFull eObj ) {
   // используется функция-член value() класса pushOnFull
   // см. раздел 11.3
   cerr << "попытка поместить значение " << eObj.value()
        << " в полный стек\n";
}
catch ( Excp ) {
   // используется функция-член print() базового класса
   Excp::print( "произошло исключение" );
}
```

остальных исключений, то они обрабатываются единообразно:

Как отмечалось в разделе 11.3, процесс поиска catch-обработчика для возбужденного исключения не похож на процесс разрешения перегрузки функций. При выборе наилучшей из устоявших функций принимаются во внимание все кандидаты, видимые в точке вызова, а при обработке исключений найденный catch-обработчик совсем не

обязательно будет лучше остальных соответствовать типу исключения. Выбирается первый подходящий обработчик, т.е. первый из просмотренных, который способен обработать данное исключение. Поэтому в списке обработчиков наиболее специализированные должны стоять ближе к началу.

Объявление исключения в `catch`-обработчике (находящееся в скобках после слова `catch`) очень похоже на объявление параметра функции. В приведенном примере оно напоминает параметр, передаваемый по значению. Объект `eObj` инициализируется копией значения объекта-исключения точно так же, как передаваемый по значению формальный параметр функции инициализируется значением фактического аргумента. Как и в случае с параметрами функции, в объявлении исключения можно использовать ссылки. Тогда `catch`-обработчик имеет доступ непосредственно к объекту-исключению, созданному выражением `throw`, а не к его локальной копии. Чтобы избежать копирования больших объектов, параметры типа класса следует объявлять как ссылки; в объявлениях исключений тоже желательно делать исключения типа класса ссылками. В зависимости от того, что находится в таком объявлении (объект или ссылка), поведение обработчика различается (мы покажем эти различия в данном разделе).

В главе 11 были введены выражения повторного возбуждения исключения, которые используются в `catch`-обработчике для передачи исключения какому-то другому обработчику выше в цепочке вызовов. Такое выражение имеет вид

```
throw;
```

Как ведет себя эта инструкция, если она расположена в catch-обработчике исключений базового класса? Например, каким будет тип повторно возбужденного исключения, если

```
void calculate( int parm ) {
   try {
      mathFunc( parm );  // возбуждает исключение divideByZero
   }
   catch ( mathExcp mExcp ) {
      // частично обрабатывает исключение
      // и генерирует объект-исключение еще раз
      throw;
   }
```

`mathFunc()` возбуждает исключение типа `divideByZero`?

```
}
```

Будет ли повторно возбужденное исключение иметь тип `divideByZero` – тот же, что и исключение, возбужденное функцией `mathFunc()`? Или тип `mathExcp`, который указан в объявлении исключения в catch-обработчике?

Напомним, что выражение `throw` повторно генерирует *исходный* объект-исключение. Так как исходный объект имеет тип `divideByZero`, то повторно возбужденное исключение будет такого же типа. В catch-обработчике объект `mExcp` инициализируется копией подобъекта объекта типа `divideByZero`, который соответствует его базовому классу `MathExcp`. Доступ к ней осуществляется только внутри catch-обработчика, она не является исходным объектом-исключением, который повторно генерируется.

Предположим, что классы в нашей иерархии исключений имеют деструкторы:

```
class pushOnFull {
public:
   pushOnFull( int i ) : _value( i ) { }
   int value() { return _value; }
   ~pushOnFull();  // вновь объявленный деструктор
private:
   int _value;
};
```

```
catch ( pushOnFull eObj ) {
   cerr << "попытка поместить значение " << eObj.value()
        << " в полный стек\n";
}
```

Когда они вызываются? Чтобы ответить на этот вопрос, рассмотрим catch-обработчик:

Поскольку в объявлении исключения `eObj` объявлен как локальный для catch-обработчика объект, а в классе `pushOnFull` есть деструктор, то `eObj` уничтожается при выходе из обработчика. Когда же вызывается деструктор для объекта-исключения, созданного в момент возбуждения исключения, – при входе в catch-обработчик или при выходе из него? Однако уничтожать исключение в любой из этих точек может быть слишком рано. Можете сказать, почему? Если catch-обработчик возбуждает исключение повторно, передавая его выше по цепочке вызовов, то уничтожать объект-исключение нельзя до момента выхода из последнего catch-обработчика.

19.2.4. Объекты-исключения и виртуальные функции

Если сгенерированный объект-исключение имеет тип производного класса, а обрабатывается catch-обработчиком для базового, то этот обработчик не может использовать особенности производного класса. Например, к функции-члену `value()`,

```
catch ( const Excp &eObj ) {
   // ошибка: в классе Excp нет функции-члена value()
   cerr << "попытка поместить значение " << eObj.value()
        << " в полный стек\n";
}
```

которая объявлена в классе `pushOnFull`, нельзя обращаться в catch-обработчике `Excp`:

Но мы можем перепроектировать иерархию классов исключений и определить виртуальные функции, которые можно вызывать из catch-обработчика для базового класса `Excp` с целью получения доступа к функциям-членам более специализированного производного:

```
// новые определения классов, включающие виртуальные функции
class Excp {
public:
   virtual void print( string msg ) {
      cerr << "Произошло исключение"
           << endl;
   }

class stackExcp : public Excp { };
class pushOnFull : public stackExcp {
public:
   virtual void print() {
      cerr << "попытка поместить значение " << _value
           << " в полный стек\n";
   }
   // ...
```

```
int main() {
   try {
      // iStack::push() возбуждает исключение pushOnFull
   } catch ( Excp eObj ) {
     eObj.print();    // хотим вызвать виртуальную функцию,
                      // но вызывается экземпляр из базового класса
   }
};
};
```

Функцию `print()` теперь можно использовать в `catch`-обработчике следующим образом:

```
}
```

Хотя возбужденное исключение имеет тип `pushOnFull`, а функция `print()` виртуальна, инструкция `eObj.print()` печатает такую строку:

```
Произошло исключение
```

Вызываемая `print()` является членом базового класса `Excp`, а не замещает ее в производном. Но почему?

Вспомните, что объявление исключения в `catch`-обработчике ведет себя почти так же, так объявление параметра. Когда управление попадает в `catch`-обработчик, то, поскольку в нем объявлен объект, а не ссылка, `eObj` инициализируется копией подобъекта `Excp` базового класса объекта исключения. Поэтому `eObj` – это объект типа `Excp`, а не `pushOnFull`. Чтобы вызвать виртуальные функции из производных классов, в объявлении исключения должен быть указатель или ссылка:

```
int main() {
   try {
      // iStack::push() возбуждает исключение pushOnFull
   } catch ( const Excp &eObj ) {
     eObj.print();   // вызывается виртуальная функция
                     // pushOnFull::print()
   }
}
```

Объявление исключения в этом примере тоже относится к базовому классу Excp, но так как eObj – ссылка и при этом именует объект-исключение типа pushOnFull, то для нее можно вызывать виртуальные функции, определенные в классе pushOnFull. Когда catch-обработчик обращается к виртуальной функции print(), вызывается функция из производного класса, и программа печатает следующую строку:

```
попытка поместить значение 879 в полный стек
```

Таким образом, ссылка в объявлении исключения позволяет вызывать виртуальные функции, ассоциированные с классом объекта-исключения.

19.2.5. Раскрутка стека и вызов деструкторов

Когда возбуждается исключение, поиск его catch-обработчика – *раскрутка стека* – начинается с функции, возбудившей исключение, и продолжается вверх по цепочке вложенных вызовов (см. раздел 11.3).

Во время раскрутки поочередно происходят аномальные выходы из просмотренных функций. Если функция захватила некоторый ресурс (например, открыла файл или выделила из хипа память), он в таком случае не освобождается.

Существует прием, позволяющий решить эту проблему. Всякий раз, когда во время поиска обработчика происходит выход из составной инструкции или блока, где определен некоторый локальный объект, для этого объекта автоматически вызывается деструктор. (Локальные объекты рассматривались в разделе 8.1.)

Например, следующий класс инкапсулирует выделение памяти для массива целых в

```
class PTR {
public:
   PTR() { ptr = new int[ chunk ]; }
   ~PTR { delete[] ptr; }
private:
   int *ptr;
};
```

конструкторе и ее освобождение в деструкторе:

Локальный объект такого типа создается в функции manip() перед вызовом mathFunc():

```
void manip( int parm ) {
   PTR localPtr;
   // ...
   mathFunc( parm );   // возбуждает исключение divideByZero
   // ...
}
```

Если mathFunc() возбуждает исключение типа divideByZero, то начинается раскрутка стека. В процессе поиска подходящего catch-обработчика проверяется и функция manip(). Поскольку вызов mathFunc() не заключен в try-блок, то manip() нужного обработчика не содержит. Поэтому стек раскручивается дальше по цепочке вызовов. Но перед выходом из manip() с необработанным исключением процесс раскрутки уничтожает все объекты типа классов, которые локальны в ней и были созданы до вызова mathFunc(). Таким образом, локальный объект localPtr уничтожается до того, как поиск пойдет дальше, а следовательно, память, на которую он указывает, будет освобождена и утечки не произойдет.

Поэтому говорят, что процесс обработки исключений в C++ поддерживает технику программирования, основной принцип которой можно сформулировать так: “захват ресурса – это инициализация; освобождение ресурса – это уничтожение”. Если ресурс реализован в виде класса и, значит, действия по его захвату сосредоточены в конструкторе, а действия по освобождению – в деструкторе (как, например, в классе PTR выше), то локальный для функции объект такого класса автоматически уничтожается при выходе из функции в результате необработанного исключения. Действия, которые должны быть выполнены для освобождения ресурса, не будут пропущены при раскрутке стека, если они инкапсулированы в деструкторы, вызываемые для локальных объектов.

Класс auto_ptr, определенный в стандартной библиотеке (см. раздел 8.4), ведет себя почти так же, как наш класс PTR. Это средство для инкапсуляции выделения памяти в конструкторе и ее освобождения в деструкторе. Если для выделения одиночного объекта из хипа используется auto_ptr, то гарантируется, что при выходе из составной инструкции или функции из-за необработанного исключения память будет освобождена.

19.2.6. Спецификации исключений

С помощью спецификации исключений (см. раздел 11.4) в объявлении функции указывается множество исключений, которые она может возбуждать прямо или косвенно. Спецификация позволяет гарантировать, что функция не возбудит не перечисленные в ней исключения.

Такую спецификацию разрешается задавать для функций-членов класса так же, как и для обычных функций; она должна следовать за списком параметров функции-члена. Например, в определении класса bad_alloc из стандартной библиотеки C++ функции-члены имеют пустую спецификацию исключений throw(), т.е. гарантированно не возбуждают никаких исключений:

```
class bad_alloc : public exception {
   // ...
public:
   bad_alloc() throw();
   bad_alloc( const bad_alloc & ) throw();
   bad_alloc & operator=( const bad_alloc & ) throw();
   virtual ~bad_alloc() throw();
   virtual const char* what() const throw();
};
```

Отметим, что если функция-член объявлена с модификатором `const` или `volatile`, как, скажем, `what()` в примере выше, то спецификация исключений должна идти после него.

Во всех объявлениях одной и той же функции спецификации исключений обязаны содержать одинаковые типы. Если речь идет о функции-члене, определение которой находится вне определения класса, то спецификации исключений в этом определении и в

```
#include <stdexcept>
// <stdexcept> определяет класс overflow_error

class transport {
   // ...
public:
   double cost( double, double ) throw ( overflow_error );
   // ...
};

// ошибка: спецификация исключений отличается от той, что задана
//         в объявлении в списке членов класса
```

объявлении функции должны совпадать:

```
double transport::cost( double rate, double distance ) { }
```

Виртуальная функция в базовом классе может иметь спецификацию исключений, отличающуюся от той, что задана для замещающей функции-члена в производном. Однако в производном классе эта спецификация для виртуальной функции должна накладывать не меньше ограничений, чем в базовом:

```
class Base {
public:
   virtual double f1( double ) throw();
   virtual int f2( int ) throw( int );
   virtual string f3() throw( int, string );
   // ...
}
class Derived : public Base {
public:
   // ошибка: спецификация исключений накладывает меньше ограничений,
   //         чем на Base::f1()
   double f1( double ) throw( string );

   // правильно: та же спецификация исключений, что и для Base::f2()
   int f2( int ) throw( int );

   // правильно: спецификация исключений f3() накладывает больше
         ограничений
   string f3( ) throw( int );
   // ...

};
```

Почему спецификация исключений в производном классе должна накладывать не меньше ограничений, чем в базовом? В этом случае мы можем быть уверены, что вызов виртуальной функции из производного класса по указателю на тип базового не нарушит

```
// гарантируется, что исключения возбуждены не будут
void compute( Base *pb ) throw()
{
   try {
      pb->f3( );  // может возбудить исключение типа int или string
   }
   // обработка исключений, возбужденных в Base::f3()
   catch ( const string & ) { }
   catch ( int ) { }
}
```

спецификацию исключений функции-члена базового класса:

Объявление `f3()` в классе `Base` гарантирует, что эта функция возбуждает лишь исключения типа `int` или `string`. Следовательно, функция `compute()` включает catch-обработчики только для них. Поскольку спецификация исключений `f3()` в производном классе `Derived` накладывает больше ограничений, чем в базовом `Base`, то при программировании в согласии с интерфейсом класса `Base` наши ожидания не будут обмануты.

В главе 11 мы говорили о том, что между типом возбужденного исключения и типом, заданным в спецификации исключений, не допускаются никакие преобразования. Однако если там указан тип класса, то функция может возбуждать исключения в виде объекта класса, открыто наследующего заданному. Аналогично, если имеется указатель на класс, то функции разрешено возбуждать исключения в виде указателя на объект класса, открыто наследующего заданному. Например:

```
class stackExcp : public Excp { };
class popObEmpty : public stackExcp { };

class pushOnFull : public stackExcp { };

void stackManip() throw( stackExcp )
{
   // ...
}
```

Спецификация исключений указывает, что `stackManip()` может возбуждать исключения не только типа `stackExcp`, но также `popOnEmpty` и `pushOnFull`. Напомним, что класс, открыто наследующий базовому, представляет собой пример отношения ЯВЛЯЕТСЯ, т.е. *является* частным случае более общего базового класса. Поскольку `popOnEmpty` и `pushOnFull` – частные случаи `stackExcp`, они не нарушают спецификации исключений функции `stackManip()`.

19.2.7. Конструкторы и функциональные try-блоки

Можно объявить функцию так, что все ее тело будет заключено в try-блок. Такие try-

```
int main() {
try {
   // тело функции main()
}
catch ( pushOnFull ) {
   // ...
}
catch ( popOnEmpty ) {
   // ...
}
```

блоки называются *функциональными*. (Мы упоминали их в разделе 11.2.) Например:

Функциональный try-блок ассоциирует группу catch-обработчиков с телом функции. Если инструкция внутри тела возбуждает исключение, то поиск его обработчика ведется среди тех, что следуют за телом функции.

Функциональный try-блок необходим для конструкторов класса. Почему? Определение

```
имя_класса( список_параметров )
// список инициализации членов:
: член1(выражение1 ) ,    // инициализация член1
  член2(выражение2 ) ,    // инициализация член2
// тело функции:
```

конструктора имеет следующий вид:

```
{ /* ... */ }
```

`выражение1` и `выражение2` могут быть выражениями любого вида, в частности функциями, которые возбуждают исключения.

Рассмотрим еще раз класс Account, описанный в главе 14. Его конструктор можно

```
inline Account::
Account( const char* name, double opening_bal )
       : _balance( opening_bal - ServiceCharge() )
{
   _name = new char[ strlen(name) + 1 ];
   strcpy( _name, name );

   _acct_nmbr = get_unique_acct_nmbr();
}
```

переопределить так:

Функция ServiceCharge(), вызываемая для инициализации члена _balance, может возбуждать исключение. Как нужно реализовать конструктор, если мы хотим обрабатывать все исключения, возбуждаемые функциями, которые вызываются при конструировании объекта типа Account?

```
inline Account::
Account( const char* name, double opening_bal )
       : _balance( opening_bal - ServiceCharge() )
{
   try {
      _name = new char[ strlen(name) + 1 ];
      strcpy( _name, name );

      _acct_nmbr = get_unique_acct_nmbr();
   }
   catch (...) {
      // специальная обработка
      // не перехватывает исключения,
      // возбужденные в списке инициализации членов
   }
}
```

Помещать try-блок в тело функции нельзя:

Поскольку try-блок не охватывает список инициализации членов, то catch-обработчик, находящийся в конце конструктора, не рассматривается при поиске кандидатов, которые способны перехватить исключение, возбужденное в функции ServiceCharge().

Использование функционального try-блока – это единственное решение, гарантирующее, что все исключения, возбужденные при создании объекта, будут перехвачены в конструкторе. Для конструктора класса Account такой try-блок можно определить следующим образом:

```
inline Account::
Account( const char* name, double opening_bal )
try
       : _balance( opening_bal - ServiceCharge() )
{
   _name = new char[ strlen(name) + 1 ];
   strcpy( _name, name );
   _acct_nmbr = get_unique_acct_nmbr();
```

```
catch (...) {
   // теперь специальная обработка
   // перехватывает исключения,
   // возбужденные в ServiceCharge()
```

```
}
}
```

Обратите внимание, что ключевое слово try находится *перед* списком инициализации членов, а составная инструкция, образующая try-блок, охватывает тело конструктора. Теперь предложение catch(...) принимается во внимание при поиске обработчика исключения, возбужденного как в списке инициализации членов, так и в теле конструктора.

19.2.8. Иерархия классов исключений в стандартной библиотеке C++

В начале этого раздела мы определили иерархию классов исключений, с помощью которой наша программа сообщает об аномальных ситуациях. В стандартной библиотеке C++ есть аналогичная иерархия, предназначенная для извещения о проблемах при выполнении функций из самой стандартной библиотеки. Эти классы исключений вы можете использовать в своих программах непосредственно или создать производные от них классы для описания собственных специфических исключений.

Корневой класс исключения в стандартной иерархии называется exception. Он определен в стандартном заголовочном файле <exception> и является базовым для всех исключений, возбуждаемых функциями из стандартной библиотеки. Класс exception

```
namespace std {
   class exception
   public:
      exception() throw();
      exception( const exception & ) throw();
      exception& operator=( const exception & ) throw();
      virtual ~exception() throw();
      virtual const char* what() const throw();
   };
```

имеет следующий интерфейс:

```
}
```

Как и всякий другой класс из стандартной библиотеки C++, exception помещен в пространство имен std, чтобы не засорять глобальное пространство имен программы.

Первые четыре функции-члена в определении класса – это конструктор по умолчанию, копирующий конструктор, копирующий оператор присваивания и деструктор. Поскольку все они открыты, любая программа может свободно создавать и копировать объекты-исключения, а также присваивать им значения. Деструктор объявлен виртуальным, чтобы сделать возможным дальнейшее наследование классу exception.

Самой интересной в этом списке является виртуальная функция what(), которая возвращает C-строку с текстовым описанием возбужденного исключения. Классы, производные от exception, могут заместить what() собственной версией, которая лучше характеризует объект-исключение.

Отметим, что все функции в определении класса exception имеют пустую спецификацию throw(), т.е. не возбуждают никаких исключений. Программа может манипулировать объектами-исключениями (к примеру, внутри catch-обработчиков типа exception), не опасаясь, что функции создания, копирования и уничтожения этих объектов возбудят исключения.

Помимо корневого exception, в стандартной библиотеке есть и другие классы, которые допустимо использовать в программе для извещения об ошибках, обычно подразделяемых на две больших категории: *логические ошибки* и *ошибки времени выполнения*.

Логические ошибки обусловлены нарушением внутренней логики программы, например логических предусловий или инвариантов класса. Предполагается, что их можно найти и предотвратить еще до начала выполнения программы. В стандартной библиотеке

```
namespace std {
   class logic_error : public exception {  // логическая ошибка
   public:
      explicit logic_error( const string &what_arg );
   };
   class invalid_argument : public logic_error { // неверный аргумент
   public:
      explicit invalid_argument( const string &what_arg );
   };
   class out_of_range : public logic_error { // вне диапазона
   public:
      explicit out_of_range( const string &what_arg );
   };
   class length_error : public logic_error { // неверная длина
   public:
      explicit length_error( const string &what_arg );
   };
   class domain_error : public logic_error { // вне допустимой области
   public:
      explicit domain_error( const string &what_arg );
   };
```

определены следующие такие ошибки:

```
}
```

Функция может возбудить исключение invalid_argument, если получит аргумент с некорректным значением; в конкретной ситуации, когда значение аргумента выходит за пределы допустимого диапазона, разрешается возбудить исключение out_of_range, а length_error используется для оповещения о попытке создать объект, длина которого превышает максимально возможную.

Ошибки времени выполнения, напротив, вызваны событием, с самой программой не связанным. Предполагается, что их нельзя обнаружить, пока программа не начала

```
namespace std {
   class runtime_error : public exception {  // ошибка времени выполнения
   public:
      explicit runtime_error( const string &what_arg );
   };
   class range_error : public runtime_error { // ошибка диапазона
   public:
      explicit range_error( const string &what_arg );
   };
   class overflow_error : public runtime_error { // переполнение
   public:
      explicit overflow_error( const string &what_arg );
   };
   class underflow_error : public runtime_error { // потеря значимости
   public:
      explicit underflow_error( const string &what_arg );
   };
```

работать. В стандартной библиотеке определены следующие такие ошибки:

```
}
```

Функция может возбудить исключение range_error, чтобы сообщить об ошибке во внутренних вычислениях. Исключение overflow_error говорит об ошибке арифметического переполнения, а underflow_error – о потере значимости.

Класс exception является базовым и для класса исключения bad_alloc, которое возбуждает оператор new(), когда ему не удается выделить запрошенный объем памяти (см. раздел 8.4), и для класса исключения bad_cast, возбуждаемого в ситуации, когда ссылочный вариант оператора dynamic_cast не может быть выполнен (см. раздел 19.1).

Переопределим оператор operator[] в шаблоне Array из раздела 16.12 так, чтобы он возбуждал исключение типа range_error, если индекс массива Array выходит за границы:

```
#include <stdexcept>
#include <string>

template <class elemType>
class Array {
public:
   // ...
   elemType& operator[]( int ix ) const
   {
      if ( ix < 0 || ix >= _size )
      {
         string eObj =
         "ошибка: вне диапазона в Array<elemType>::operator[]()";

         throw out_of_range( eObj );
      }
      return _ia[ix];
   }

   // ...
private:
   int _size;
   elemType *_ia;
};
```

Для использования предопределенных классов исключений в программу необходимо включить заголовочный файл <stdexcept>. Описание возбужденного исключения содержится в объекте eObj типа string. Эту информацию можно извлечь в обработчике

```
int main()
{
   try {
      // функция main() такая же, как в разделе 16.2
   }
   catch ( const out_of_range &excep ) {
      // печатается:
      // ошибка: вне диапазона в Array<elemType>::operator[]()
      cerr << excep.what() << "\n";
      return -1;
   }
}
```

с помощью функции-члена what():

В данной реализации выход индекса за пределы массива в функции try_array() приводит к тому, что оператор взятия индекса operator[]() класса Array возбуждает исключение типа out_of_range, которое перехватывается в main().

Упражнение 19.5

Какие исключения могут возбуждать следующие функции:

```
#include <stdexcept>


(a) void operate() throw( logic_error );

(b) int mathErr( int ) throw( underflow_error, overflow_error );

(c) char manip( string ) throw( );
```

Упражнение 19.6

Объясните, как механизм обработки исключений в C++ поддерживает технику программирования “захват ресурса – это инициализация; освобождение ресурса – это уничтожение”.

Упражнение 19.7

```
#include <stdexcept>

int main() {
   try {
      // использование функций из стандартной библиотеки
   }
   catch( exception ) {
   }
   catch( runtime_error &re ) {
   }
   catch( overflow_error eobj ) {
   }
```

Исправьте ошибку в списке catch-обработчиков для данного try-блока:

```
}
```

Упражнение 19.8

```
int main() {
   // использование стандартной библиотеки
```

Дана программа на C++:

```
}
```

Модифицируйте `main()` так, чтобы она перехватывала все исключения, возбуждаемые функциями стандартной библиотеки. Обработчики должны печатать сообщение об ошибке, ассоциированное с исключением, а затем вызывать функцию `abort()` (она определена в заголовочном файле `<cstdlib>`) для завершения `main()`.

19.3. Разрешение перегрузки и наследование A

Наследование классов оказывает влияние на все аспекты разрешения перегрузки функций (см. раздел 9.2). Напомним, что эта процедура состоит из трех шагов:

1. Отбор функций-кандидатов.

2. Отбор устоявших функций.
3. Выбор наилучшей из устоявших функции.

Отбор функций-кандидатов зависит от наследования потому, что на этом шаге принимаются во внимание функции, ассоциированные с базовыми классами, – как их функции-члены, так и функции, объявленные в тех же пространствах имен, где определены базовые классы. Отбор устоявших функций также зависит от наследования, поскольку множество преобразований формальных параметров функции в фактические аргументы расширяется пользовательскими преобразованиями. Кроме того, наследование оказывает влияние на ранжирование последовательностей трансформаций аргументов, а значит, и на выбор наилучшей из устоявших функции. В данном разделе мы рассмотрим влияние наследования на эти три шага разрешения перегрузки более подробно.

19.3.1. Функции-кандидаты

Наследование влияет на первый шаг процедуры разрешения перегрузки функции – формирование множества кандидатов для данного вызова, причем это влияние может быть различным в зависимости от того, рассматривается ли вызов обычной функции вида

```
func( args );
```

```
object.memfunc( args );
```

или функции-члена с помощью операторов доступа “точка” или “стрелка”:

```
pointer->memfunc( args );
```

В данном разделе мы изучим оба случая.

Если аргумент обычной функции имеет тип класса, ссылки или указателя на тип класса, и класс определен в пространстве имен, то кандидатами будут все одноименные функции, объявленные в этом пространстве, даже если они невидимы в точке вызова (подробнее об этом говорилось в разделе 15.10). Если аргумент при наследовании имеет тип класса, ссылки или указателя на тип класса, и у этого класса есть базовые, то в множество кандидатов добавляются также функции, объявленные в тех пространствах имен, где

```
namespace NS {
   class ZooAnimal { /* ... */ };
   void display( const ZooAnimal& );
}

// базовый класс Bear объявлен в пространстве имен NS
class Bear : public NS::ZooAnimal { };

int main() {
   Bear baloo;

   display( baloo );
   return 0;
}
```

определены базовые классы. Например:

Аргумент baloo имеет тип класса Bear. Кандидатами для вызова display() будут не только функции, объявления которых видимы в точке ее вызова, но также и те, что объявлены в пространствах имен, в которых объявлены класс Bear и его базовый класс ZooAnimal. Поэтому в множество кандидатов добавляется функция display(const ZooAnimal&), объявленная в пространстве имен NS.

Если аргумент имеет тип класса и в определении этого класса объявлены функции-друзья с тем же именем, что и вызванная функция, то эти друзья также будут кандидатами, даже если их объявления не видны в точке вызова (см. раздел 15.10). Если аргумент при наследовании имеет тип класса, у которого есть базовые, то в множество кандидатов добавляются одноименные функции-друзья каждого из них. Предположим, что в

```
namespace NS {
   class ZooAnimal {
      friend void display( const ZooAnimal& );
   };
}

// базовый класс Bear объявлен в пространстве имен NS
class Bear : public NS::ZooAnimal { };

int main() {
   Bear baloo;

   display( baloo );
   return 0;
}
```

предыдущем примере display() объявлена как функция-друг ZooAnimal:

Аргумент baloo функции display() имеет тип Bear. В его базовом классе ZooAnimal функция display() объявлена другом, поэтому она является членом пространства имен NS, хотя явно в нем не объявлена. При обычном просмотре NS она не была бы найдена. Однако поскольку аргумент display() имеет тип Bear, то объявленная в ZooAnimal функция-друг добавляется в множество кандидатов.

Таким образом, если при вызове обычной функции задан аргумент, который представляет собой объект класса, ссылку или указатель на объект класса, то множество функций-кандидатов является объединением следующих множеств:

- функций, видимых в точке вызова;
- функций, объявленных в тех пространствах имен, где определен тип класса или любой из его базовых;
- функций, являющихся друзьями этого класса или любого из его базовых.

Наследование влияет также на построение множества кандидатов для вызова функции-члена с помощью операторов “точка” или “стрелка”. В разделе 18.4 мы говорили, что объявление функции-члена в производном классе не перегружает, а скрывает одноименные функции-члены в базовом, даже если их списки параметров различны:

```
class ZooAnimal {
public:
   Time feeding_time( string );
   // ...
};
class Bear : public ZooAnimal {
public:
   // скрывает ZooAnimal::feeding_time( string )
   Time feeding_time( int );
   // ...
};

Bear Winnie;

// ошибка: ZooAnimal::feeding_time( string ) скрыта
Winnie.feeding_time( "Winnie" );
```

Функция-член `feeding_time(int)`, объявленная в классе `Bear`, скрывает `feeding_time(string)`, объявленную в `ZooAnimal`, базовом для `Bear`. Поскольку функция-член вызывается через объект `Winnie` типа `Bear`, то при поиске кандидатов для этого вызова просматривается только область видимости класса `Bear`, и единственным кандидатом будет `feeding_time(int)`. Так как других кандидатов нет, вызов считается ошибочным.

Чтобы исправить ситуацию и заставить компилятор считать одноименные функции-члены базового и производного классов перегруженными, разработчик производного класса может ввести функции-члены базового класса в область видимости производного

```
class Bear : public ZooAnimal {
public:
   // feeding_time( int ) перегружает экземпляр из класса ZooAnimal
   using ZooAnimal::feeding_time;
   Time feeding_time( int );
   // ...
};
```

с помощью using-объявлений:

Теперь обе функции `feeding_time()` находятся в области видимости класса `Bear` и,

```
// правильно: вызывается ZooAnimal::feeding_time( string )
Winnie.feeding_time( "Winnie" );
```

следовательно, войдут в множество кандидатов:

В такой ситуации вызывается функция-член `feeding_time( string )`.

В случае множественного наследования при формировании совокупности кандидатов объявления функций-членов должны быть найдены в одном и том же базовом классе, иначе вызов считается ошибочным. Например:

```
class Endangered {
public:
   ostream& print( ostream& );
   // ...
{;

class Bear : public( ZooAnimal ) {
public:
   void print( );
   using ZooAnimal::feeding_time;
   Time feeding_time( int );
   // ...
};

class Panda : public Bear, public Endangered {
public:
   // ...
};

int main()
{
   Panda yin_yang;

   // ошибка: неоднозначность: одна из
   //         Bear::print()
   //         Endangered::print( ostream& )
   yin_yang.print( cout );

   // правильно: вызывается Bear::feeding_time()
   yin_yang.feeding_time( 56 );

}
```

При поиске объявления функции-члена `print()` в области видимости класса `Panda` будут найдены как `Bear::print()`, так и `Endangered::print()`. Поскольку они не находятся в одном и том же базовом классе, то даже при разных списках параметров этих функций множество кандидатов оказывается пустым и вызов считается ошибочным. Для исправления ошибки в классе `Panda` следует определить собственную функцию `print()`. При поиске объявления функции-члена `feeding_time()` в области видимости `Panda` будут найдены `ZooAnimal::feeding_time()` и `Bear::feeding_time()` – они расположены в области видимости класса `Bear`. Так как эти объявления найдены в одном и том же базовом классе, множество кандидатов для данного вызова включает обе функции, а выбирается `Bear::feeding_time()`.

19.3.2. Устоявшие функции и последовательности пользовательских преобразований

Наследование оказывает влияние и на второй шаг разрешения перегрузки функции: отбор устоявших из множества кандидатов. Устоявшей называется функция, для которой существуют приведения типа каждого фактического аргумента к типу соответственного формального параметра.

В разделе 15.9 мы показали, как разработчик класса может предоставить пользовательские преобразования для объектов этого класса, которые неявно вызываются компилятором для трансформации фактического аргумента функции в тип соответственного формального параметра. Пользовательские преобразования бывают двух видов: конвертер или конструктор с одним параметром без ключевого слова

explicit. При наследовании на втором шаге разрешения перегрузки рассматривается более широкое множество таких преобразований.

Конвертеры наследуются, как и любые другие функции-члены класса. Например, мы

```
class ZooAnimal {
public:

   // конвертер: ZooAnimal ==> const char*
   operator const char*();

   // ...
```

можем написать следующий конвертер для ZooAnimal:

```
};
```

Производный класс Bear наследует его от своего базового ZooAnimal. Если значение типа Bear используется в контексте, где ожидается const char*, то неявно вызывается

```
extern void display( const char* );

Bear yogi;

// правильно: yogi ==> const char*
```

конвертер для преобразования Bear в const char*:

```
display( yogi );
```

Конструкторы с одним аргументом без ключевого слова explicit образуют другое множество неявных преобразований: из типа параметра в тип своего класса. Определим

```
class ZooAnimal {
public:
   // преобразование: int ==> ZooAnimal
   ZooAnimal( int );

   // ...
```

такой конструктор для ZooAnimal:

```
};
```

Его можно использовать для приведения значения типа int к типу ZooAnimal. Однако конструкторы не наследуются. Конструктор ZooAnimal нельзя применять для

```
const int cageNumber = 8788l

void mumble( const Bear & );

// ошибка: ZooAnimal( int ) не используется
```

преобразования объекта в случае, когда целевым является тип производного класса:

```
mumble( cageNumber );
```

Поскольку целевым типом является Bear – тип параметра функции mumble(), то рассматриваются только его конструкторы.

19.3.3. Наилучшая из устоявших функций

Наследование влияет и на третий шаг разрешения перегрузки – выбор наилучшей из устоявших функций. На этом шаге ранжируются преобразования типов, с помощью которых можно привести фактические аргументы функции к типам соответственных формальных параметров. Следующие неявные преобразования имеют тот же ранг, что и стандартные (стандартные преобразования рассматривались в разделе 9.3):

- преобразование аргумента типа производного класса в параметр типа любого из его базовых;
- преобразование указателя на тип производного класса в указатель на тип любого из его базовых;
- инициализация ссылки на тип базового класса с помощью l-значения типа производного.

Они не являются пользовательскими, так как не зависят от конвертеров и конструкторов,

```
extern void release( const ZooAnimal& );
Panda yinYang;

// стандартное преобразование: Panda -> ZooAnimal
```

имеющихся в классе:

```
release( yinYang );
```

Поскольку аргумент yinYang типа Panda инициализирует ссылку на тип базового класса, то преобразование имеет ранг стандартного.

В разделе 15.10 мы говорили, что стандартные преобразования имеют более высокий

```
class Panda : public Bear,
              public Endangered
{
   // наследует ZooAnimal::operator const char *()
};

Panda yinYang;

extern void release( const ZooAnimal& );
extern void release( const char * );

// стандартное преобразование: Panda -> ZooAnimal
// выбирается: release( const ZooAnimal& )
```

ранг, чем пользовательские:

```
release( yinYang );
```

Как release(const char*), так и release(ZooAnimal&) являются устоявшими функциями: первая потому, что инициализация параметра-ссылки значением аргумента – стандартное преобразование, а вторая потому, что аргумент можно привести к типу

const char* с помощью конвертера ZooAnimal::operator const char*(), который представляет собой пользовательское преобразование. Так как стандартное преобразование лучше пользовательского, то в качестве наилучшей из устоявших выбирается функция release(const ZooAnimal&).

При ранжировании различных стандартных преобразований из производного класса в базовые лучшим считается приведение к тому базовому классу, который ближе к производному. Так, показанный ниже вызов не будет неоднозначным, хотя в обоих случаях требуется стандартное преобразование. Приведение к базовому классу Bear лучше, чем к ZooAnimal, поскольку Bear ближе к классу Panda. Поэтому лучшей из

```
extern void release( const ZooAnimal& );
extern void release( const Bear& );

// правильно: release( const Bear& )
```

устоявших будет функция release(const Bear&):

```
release( yinYang );
```

Аналогичное правило применимо и к указателям. При ранжировании стандартных преобразований из указателя на тип производного класса в указатели на типы различных базовых лучшим считается то, для которого базовый класс наименее удален от производного. Это правило распространяется и на тип void*.

Стандартное преобразование в указатель на тип любого базового класса всегда лучше,

```
void receive( void* );
```

чем преобразование в void*. Например, если дана пара перегруженных функций:

```
void receive( ZooAnimal* );
```

то наилучшей из устоявших для вызова с аргументом типа Panda* будет receive(ZooAnimal*).

В случае множественного наследования два стандартных преобразования из типа производного класса в разные типы базовых могут иметь одинаковый ранг, если оба базовых класса равноудалены от производного. Например, Panda наследует классам Bear и Endangered. Поскольку они равноудалены от производного Panda, то преобразования объекта Panda в любой из этих классов одинаково хороши. Но тогда единственной наилучшей из устоявших функции для следующего вызова не существует, и он считается

```
extern void mumble( const Bear& );
extern void mumble( const Endangered& );

/* ошибка: неоднозначный вызов:
 * может быть выбрана любая из двух функций
 * void mumble( const Bear& );
* void mumble( const Endangered& );
*/
```

ошибочным:

```
mumble( yinYang );
```

Для разрешения неоднозначности программист может применить явное приведение типа:

```
mumble( static_cast< Bear >( yinYang ) );  // правильно
```

Инициализация объекта производного класса или ссылки на него объектом типа базового, а также преобразование указателя на тип базового класса в указатель на тип производного никогда не выполняются компилятором неявно. (Однако их можно выполнить с помощью явного применения dynamic_cast, как мы видели в разделе 19.1.) Для данного вызова не существует наилучшей из устоявших функции, так как нет

```
extern void release( const Bear& );
extern void release( const Panda& );

ZooAnimal za;

// ошибка: нет соответствия
```

неявного преобразования аргумента типа ZooAnimal в тип производного класса:

```
release( za );
```

В следующем примере наилучшей из устоявших будет release(const char*). Это может показаться удивительным, так как к аргументу применена последовательность пользовательских преобразований, в которой участвует конвертер const char*(). Но поскольку неявного приведения от типа базового класса к типу производного не существует, то release(const Bear&) не является устоявшей функцией, так что

```
Class ZooAnimal {
public:
   // преобразование: ZooAnimal ==> const char*
   operator const char*();

   // ...
};

extern void release( const char* );
extern void release( const Bear& );

ZooAnimal za;

// za ==> const char*
// правильно: release( const char* )
```

остается только release(const char*):

```
release( za );
```

Упражнение 19.9

Дана такая иерархия классов:

```
class Base1 {
public:
   ostream& print();
   void debug();
   void writeOn();
   void log( string );
   void reset( void *);
   // ...
};

class Base2 {
public:
   void debug();
   void readOn();
   void log( double );
   // ...
};

class MI : public Base1, public Base2 {
public:
   ostream& print();
   using Base1::reset;
   void reset( char * );
   using Base2::log;
   using Base2::log;
   // ...
};
```

```
MI *pi = new MI;
(a) pi->print();   (c) pi->readOn();   (e) pi->log( num );
```

Какие функции входят в множество кандидатов для каждого из следующих вызовов:

```
(b) pi->debug();   (d) pi->reset(0);   (f) pi->writeOn();
```

Упражнение 19.10

```
class Base {
public:
   operator int();
   operator const char *();
   // ...
};
class Derived : public Base {
public:
   operator double();
   // ...
};
```

Дана такая иерархия классов:

Удастся ли выбрать наилучшую из устоявших функций для каждого из следующих вызовов? Назовите кандидаты, устоявшие функции и преобразования типов аргументов для каждой из них, наилучшую из устоявших (если она есть):

```
(a) void operate( double );
    void operate( string );
    void operate( const Base & );

    Derived *pd = new Derived;
```

```
(b) void calc( int );
    void calc( double );
    void calc( const Derived & );

    Base *pb = new Derived;

    operate( *pd );

    operate( *pb );
```

20

20. Библиотека iostream

Частью стандартной библиотеки C++ является *библиотека iostream* – объектно-ориентированная иерархия классов, где используется и множественное, и виртуальное наследование. В ней реализована поддержка для файлового ввода/вывода данных встроенных типов. Кроме того, разработчики классов могут расширять эту библиотеку для чтения и записи новых типов данных.

Для использования библиотеки `iostream` в программе необходимо включить заголовочный файл

```
#include <iostream>
```

Операции ввода/вывода выполняются с помощью классов `istream` (потоковый ввод) и `ostream` (потоковый вывод). Третий класс, `iostream`, является производным от них и поддерживает двунаправленный ввод/вывод. Для удобства в библиотеке определены три стандартных объекта-потока:

- `cin` – объект класса `istream`, соответствующий *стандартному вводу*. В общем случае он позволяет читать данные с терминала пользователя;
- `cout` – объект класса `ostream`, соответствующий *стандартному выводу*. В общем случае он позволяет выводить данные на терминал пользователя;
- `cerr` – объект класса `ostream`, соответствующий *стандартному выводу для ошибок*. В этот поток мы направляем сообщения об ошибках программы.

Вывод осуществляется, как правило, с помощью перегруженного оператора сдвига влево (<<), а ввод – с помощью оператора сдвига вправо (>>):

```
#include <iostream>
#include <string>

int main()
{
   string in_string;

   // вывести литерал на терминал пользователя
   cout << "Введите свое имя, пожалуйста: ";

   // прочитать ответ пользователя в in_string
   cin >> in_string;

   if ( in_string.empty() )
      // вывести сообщение об ошибке на терминал пользователя
      cerr << "ошибка: введенная строка пуста!\n";
   else cout << "Привет, " << in_string << "!\n";

}
```

Назначение операторов легче запомнить, если считать, что каждый “указывает” в сторону перемещения данных. Например,

```
>> x
```

перемещает данные *в* x, а

```
<< x
```

перемещает данные *из* x. (В разделе 20.1 мы покажем, как библиотека iostream поддерживает ввод данных, а в разделе 20.5 – как расширить ее для ввода данных новых типов. Аналогично раздел 20.2 посвящен поддержке вывода, а раздел 20.4 – расширению для вывода данных определенных пользователем типов.)

Помимо чтения с терминала и записи на него, библиотека iostream поддерживает чтение и запись в файлы. Для этого предназначены следующие классы:

- ifstream, производный от istream, связывает ввод программы с файлом;
- ofstream, производный от ostream, связывает вывод программы с файлом;
- fstream, производный от iostream, связывает как ввод, так и вывод программы с файлом.

Чтобы использовать часть библиотеки iostream, связанную с файловым вводом/выводом, необходимо включить в программу заголовочный файл

```
#include <fstream>
```

(Файл fstream уже включает iostream, так что включать оба файла необязательно.) Файловый ввод/вывод поддерживается теми же операторами:

```
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>
int main()
{
   string ifile;

   cout << "Введите имя файла для сортировки: ";
   cin >> ifile;

   // сконструировать объект класса ifstream для ввода из файла
   ifstream infile( ifile.c_str() );

   if ( ! infile ) {
      cerr << "ошибка: не могу открыть входной файл: "
           << ifile << endl;
      return -1;
   }

   string ofile = ifile + ".sort";

   // сконструировать объект класса ofstream для вывода в файл
   ofstream outfile( ofile.c_str() );
   if ( ! outfile) {
      cerr << "ошибка: не могу открыть выходной файл: "
           << ofile << endl;
      return -2;
   }

   string buffer;
   vector< string, allocator > text;

   int cnt = 1;
   while ( infile >> buffer ) {
         text.push_back( buffer );
         cout << buffer << (cnt++ % 8 ? " " : "\n" );
   }

   sort( text.begin(), text.end() );

   // выводим отсортированное множество слов в файл
   vector< string >::iterator iter = text.begin();
   for ( cnt = 1; iter != text.end(); ++iter, ++cnt )
       outfile << *iter
               << (cnt % 8 ? " " : "\n" );

   return 0;
}
```

Вот пример сеанса работы с этой программой. Нас просят ввести файл для сортировки. Мы набираем **alice_emma** (набранные на клавиатуре символы напечатаны полужирным шрифтом). Затем программа направляет на стандартный вывод все, что прочитала из файла:

```
Введите имя файла для сортировки: alice_emma
Alice Emma has long flowing red hair. Her
Daddy says when the wind blows through her
hair, it looks almost alive, like a fiery
bird in flight. A beautiful fiery bird, he
tells her, magical but untamed. "Daddy, shush, there
```

```
is no such creature," she tells him, at
the same time wanting him to tell her
more. Shyly, she asks, "I mean, Daddy, is
there?"
```

Далее программа выводит в файл `outfile` отсортированную последовательность строк. Конечно, на порядок слов влияют знаки препинания; в следующем разделе мы это исправим:

```
"Daddy, "I A Alice Daddy Daddy, Emma Her
Shyly, a alive, almost asks, at beautiful bird
bird, blows but creature," fiery fiery flight. flowing
hair, hair. has he her her her, him
him, in is is it like long looks
magical mean, more. no red same says she
she shush, such tell tells tells the the
there there?" through time to untamed. wanting when
wind
```

(В разделе 20.6 мы познакомимся с файловым вводом/выводом более подробно.)

Библиотека `iostream` поддерживает также ввод/вывод в область памяти, при этом поток связывается со строкой в памяти программы. С помощью потоковых операторов ввода/вывода мы можем записывать данные в эту строку и читать их оттуда. Объект для строкового ввода/вывода определяется как экземпляр одного из следующих классов:

- `istringstream`, производный от `istream`, читает из строки;
- `ostringstream`, производный от `ostream`, пишет в строку;
- `stringstream`, производный от `iostream`, выполняет как чтение, так и запись.

Для использования любого из этих классов в программу нужно включить заголовочный файл

```
#include <sstream>
```

(Файл `sstream` уже включает `iostream`, так что включать оба файла необязательно.) В следующем фрагменте объект класса `ostringstream` используется для форматирования сообщения об ошибке, которое возвращается вызывающей программе.

```
#include <sstream>

string program_name( "our_program" );
string version( 0.01 );

// ...

string mumble( int *array, int size )
{
   if ( ! array ) {
      ostringstream out_message;

      out_message << "ошибка: "
                  << program_name << "--" << version
                  << ": " << __FILE__ << ": " << __LINE__
                  << " -- указатель равен 0; "
                  << " а должен адресовать массив.\n";

      // возвращаем строку, в которой находится сообщение
      return out_message.str();
   }
   // ...
}
```

(В разделе 20.8 мы познакомимся со строковым вводом/выводом более подробно.)

Потоки ввода/вывода поддерживают два предопределенных типа: char и wchar_t. В этой главе мы расскажем только о чтении и записи в потоки данных типа char. Помимо них, в библиотеке iostream имеется набор классов и объектов для работы с типом wchar_t. Они отличаются от соответствующих классов, использующих тип char, наличием префикса 'w'. Так, объект стандартного ввода называется wcin, стандартного вывода – wcout, стандартного вывода для ошибок – wcerr. Но набор заголовочных файлов для char и wchar_t один и тот же.

Классы для ввода/вывода данных типа wchar_t называются wostream, wistream, wiostream, для файлового ввода/вывода – wofstream, wifstream, wfstream, а для строкового – wostringstream, wistringstream, wstringstream.

20.1. Оператор вывода <<

Оператор вывода обычно применяется для записи на стандартный вывод cout.

```
#include <iostream>

int main()
{
    cout << "сплетница Анна Ливия\n";
}
```

Например, программа

печатает на терминале строку:

```
сплетница Анна Ливия
```

Имеются операторы, принимающие аргументы любого встроенного типа данных, включая `const char*`, а также типов `string` и `complex` из стандартной библиотеки. Любое выражение, включая вызов функции, может быть аргументом оператора вывода при условии, что результатом его вычисления будет тип, принимаемый каким-либо

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

int main()
{
    cout << "Длина 'Улисс' равна:\t";
    cout << strlen( "Улисс" );
    cout << '\n';
    cout << "Размер 'Улисс' равен:\t";
    cout << sizeof( "Улисс" );
    cout << endl;
}
```

вариантом этого оператора. Например, программа

выводит на терминал следующее:

```
Длина 'Улисс' равна:7
Размер 'Улисс' равен:8
```

`endl` – это *манипулятор* вывода, который вставляет в выходной поток символ перехода на новую строку, а затем сбрасывает буфер объекта `ostream`. (С буферизацией мы познакомимся в разделе 20.9.)

Операторы вывода, как правило, удобнее сцеплять в одну инструкцию. Например,

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

int main()
{
    // операторы вывода можно сцеплять

    cout << "Длина 'Улисс' равна:\t";
         << strlen( "Улисс" ) << '\n';

    cout << "Размер 'Улисс' равен:\t"
         << sizeof( "Улисс" ) << endl;
}
```

предыдущую программу можно записать таким образом:

Сцепление операторов вывода (и ввода тоже) возможно потому, что результатом выражения

```
cout << "некоторая строка";
```

служит левый операнд оператора вывода, т.е. сам объект cout. Затем этот же объект передается следующему оператору и далее по цепочке (мы говорим, что оператор << левоассоциативен).

Имеется также предопределенный оператор вывода для указательных типов, который печатает адрес объекта. По умолчанию адреса отображаются в шестнадцатеричном виде.

```
#include <iostream>

int main()
{
    int i = 1024;
    int *pi = &i;

    cout << "i:   " << i
         << "\t&i:\t" << &i << '\n';

    cout << "*pi: "      << *pi
         << "\tpi:\t"    << pi  << endl
         << "\t\t&pi:\t" << &pi << endl;
}
```

Например, программа

выводит на терминал следующее:

```
i:   1024  &i: 0x7fff0b4
*pi: 1024  pi: 0x7fff0b4
          &pi: 0x7fff0b0
```

Позже мы покажем, как напечатать адреса в десятичном виде.

Следующая программа ведет себя странно. Мы хотим напечатать адрес, хранящийся в

```
#include <iostream>

const char *str = "vermeer";
int main()
{
   const char *pstr = str;
   cout << "Адрес pstr равен: "
        << pstr << endl;
}
```

переменной pstr:

Но после компиляции и запуска программа неожиданно выдает такую строку:

```
Адрес pstr равен: vermeer
```

Проблема в том, что тип const char* интерпретируется как С-строка. Чтобы все же напечатать адрес, хранящийся в pstr, необходимо подавить обработку типа const

char* по умолчанию. Для этого мы сначала убираем спецификатор const, а затем приводим pstr к типу void*:

```
<< static_cast<void*>(const_cast<char*>(pstr))
```

Теперь программа выводит ожидаемый результат:

```
Адрес pstr равен: 0x116e8
```

```
#include <iostream>

inline void
max_out( int val1, int val2 )
{
   cout << ( val1 > val2 ) ? val1 : val2;
}

int main()
{
    int ix = 10, jx = 20;

    cout << "Большее из " << ix
         << ", " << jx << " равно ";

    max_out( ix, jx );

    cout << endl;
}
```

А вот еще одна загадка. Нужно напечатать большее из двух чисел:

Однако программа выдает неправильный результат:

```
Большее из 10, 20 равно 0
```

Проблема в том, что оператор вывода имеет более высокий приоритет, чем оператор условного выражения, поэтому печатается результат сравнения val1 и val2. Иными словами, выражение

```
cout << ( val1 > val2 ) ? val1 : val2;
```

вычисляется как

```
(cout << ( val1 > val2 )) ? val1 : val2;
```

Поскольку val1 не больше val2, то результатом сравнения будет false, обозначаемый нулем. Чтобы изменить приоритет операций, весь оператор условного выражения следует заключить в скобки:

```
cout << ( val1 > val2  ? val1 : val2 );
```

Теперь результат получается правильный:

```
Большее из 10, 20 равно 20
```

Такого рода ошибку было бы проще найти, если бы значения литералов `true` и `false` типа `bool` печатались как строки, а не как `1` и `0`. Тогда мы увидели бы строку:

```
Большее из 10, 20 равно false
```

и все стало бы ясно. По умолчанию литерал `false` печатается как `0`, а `true` – как `1`. Это можно изменить, воспользовавшись манипулятором `boolalpha()`, что и сделано в

```
int main()
{
          cout << "печать значений типа bool по умолчанию: "
               << true << " " << false
               << "\nи в виде строк: "
               << boolalpha()
               << true << " " << false
               << endl;
}
```

следующей программе:

Вот результат:

```
печать значений типа bool по умолчанию: 1 0
и в виде строк: true false
```

Для вывода массива, а также вектора или отображения, необходимо обойти все элементы

```
#include <iostream>
#include <vector>
#include <string>

string pooh_pals[] = {
          "Тигра", "Пятачок", "Иа-Иа", "Кролик"
};

int main()
{
          vector<string> ppals( pooh_pals, pooh_pals+4 );

     vector<string>::iterator iter = ppals.begin();
     vector<string>::iterator iter_end = ppals.end();

          cout << "Это друзья Пуха: ";
          for ( ; iter != iter_end; iter++ )
                cout << *iter << " ";

          cout << endl;
```

и напечатать каждый из них:

```
}
```

Вместо того чтобы явно обходить все элементы контейнера, выводя каждый по очереди, можно воспользоваться потоковым итератором ostream_iterator. Так выглядит эквивалентная программа, где используется эта техника (подробное обсуждение

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>

string pooh_pals[] = {
          "Тигра", "Пятачок", "Иа-Иа", "Кролик"
};

int main()
{
          vector<string> ppals( pooh_pals, pooh_pals+4 );

     vector<string>::iterator iter = ppals.begin();
     vector<string>::iterator iter_end = ppals.end();

          cout << "Это друзья Пуха: ";

     // копируем каждый элемент в cout ...
          ostream_iterator< string > output( cout, " " );
          copy( iter, iter_end, output );

          cout << endl;
```

итератора ostream_iterator см. в разделе 12.4):

```
}
```

Программа печатает такую строку:

```
Ýòî äðóçüÿ Ïóõà: Òèãðà Ïÿòà÷îê Èà-Èà Êðîëèê
```

Упражнение 20.1

```
string sa[4] = { "пух", "тигра", "пятачок", "иа-иа" };
vector< string > svec( sa, sa+4 );
string robin( "кристофер робин" );
const char *pc = robin.c_str();
int ival = 1024;
char blank = ' ';
double dval = 3.14159;
```

Даны следующие определения объектов:

```
complex purei( 0, 7 );
```

(a) Направьте значение каждого объекта в стандартный вывод.

(b) Напечатайте значение адреса pc.

(c) Напечатайте наименьшее из двух значений `ival` и `dval`, пользуясь оператором условного выражения:

```
ival < dval ? ival : dval
```

20.2. Ввод

Основное средство реализации ввода – это оператор сдвига вправо (`>>`). Например, в следующей программе из стандартного ввода читается последовательность значений типа

```
#include <iostream>
#include <vector>

int main()
{
   vector<int> ivec;
   int ival;

   while ( cin >> ival )
           ivec.push_back( ival );
   // ...
```

`int` и помещается в вектор:

```
}
```

Подвыражение

```
cin >> ival;
```

читает целое число из стандартного ввода и копирует его в переменную `ival`. Результатом является левый операнд – объект класса `istream`, в данном случае `cin`. (Как мы увидим, это позволяет сцеплять операторы ввода.)

Выражение

```
while ( cin >> ival )
```

читает последовательность значений, пока `cin` не станет равно `false`. Значение `istream` может быть равно `false` в двух случаях: достигнут конец файла (т.е. все значения из файла прочитаны успешно) или встретилось неверное значение, скажем 3.14159 (десятичная точка недопустима в целом числе), `1e-1` (буква `e` недопустима) или любой строковый литерал. Если вводится неверное значение, объект `istream` переводится в состояние ошибки и чтение прекращается. (В разделе 20.7 мы подробнее расскажем о таких состояниях.)

Есть набор предопределенных операторов ввода, принимающих аргументы любого встроенного типа, включая С-строки, а также стандартных библиотечных типов `string` и `complex`:

```
#include <iostream>
#include <string>

int main()
{
    int item_number;
    string item_name;
    double item_price;

    cout << "Пожалуйста, введите item_number, item_name и price: "
         << endl;

    cin >> item_number;
    cin >> item_name;
    cin >> item_price;

    cout << "Введены значения: item# "
           << item_number << " "
           << item_name << " @$"
           << item_price << endl;

}
```

Вот пример выполнения этой программы:

```
Пожалуйста, введите item_number, item_name и price:
10247 widget 19.99
Введены значения: item# 10247 widget @$19.99
```

Можно ввести каждый элемент на отдельной строке. По умолчанию оператор ввода отбрасывает все разделяющие пустые символы: пробел, символ табуляции, символ перехода на новую строку, символ перевода страницы и символ возврата каретки. (О том, как отменить это поведение, см. в разделе 20.9.)

```
Пожалуйста, введите item_number, item_name и price:
10247
widget
19.99
Введены значения: item# 10247 widget @$19.99
```

При чтении ошибка `iostream` более вероятна, чем при записи. Если мы вводим такую последовательность:

```
// ошибка: item_name должно быть вторым
BuzzLightyear 10009 8.99
```

то инструкция

```
cin >> item_number;
```

закончится ошибкой ввода, поскольку `BuzzLightyear` не принадлежит типу `int`. При проверке объекта `istream` будет возвращено `false`, поскольку возникло состояние ошибки. Более устойчивая к ошибкам реализация выглядит так:

```
cin >> item_number;
if ( ! cin )
    cerr << "ошибка: введено некорректное значение item_number!\n";
```

Хотя сцепление операторов ввода поддерживается, проверить корректность каждой отдельной операции нельзя, поэтому пользоваться таким приемом следует лишь тогда,

```
#include <iostream>
#include <string>

int main()
{
    int item_number;
    string item_name;
    double item_price;

    cout << "Пожалуйста, введите item_number, item_name и price: "
         << endl;

    // хорошо, но легче допустить ошибку
    cin >> item_number >> item_name >> item_price;

    cout << "Введены значения: item# "
              << item_number << " "
              << item_name << " @$"
              << item_price << endl;
}
```

когда ошибка невозможна. Наша программа теперь выглядит так:

Последовательность

```
ab c
d    e
```

составлена из девяти символов: `'a'`, `'b'`, `' '` (пробел), `'c'`, `'\n'` (переход на новую строку), `'d'`, `'\t'` (табуляция), `'e'` и `'\n'`. Однако приведенная программа читает лишь

```
#include <iostream>

int main()
{
   char ch;

   // прочитать и вывести каждый символ
   while ( cin >> ch )
           cout << ch;
   cout << endl;

   // ...
}
```

пять букв:

И печатает следующее:

```
abcde
```

По умолчанию все пустые символы отбрасываются. Если нам нужны и они, например для сохранения формата входного текста или обработки пустых символов (скажем, для подсчета количества символов перехода на новую строку), то можно воспользоваться функцией-членом `get()` класса `istream` (обычно в паре с ней употребляется функция-

```
#include <iostream>

int main()
{
   char ch;

   // читать все символы, в том числе пробельные
   while ( cin.get( ch ))
           cout.put( ch );
   // ...
}
```

член `put()` класса `ostream`; они будут рассмотрены ниже). Например:

Другая возможность сделать это – использовать манипулятор `noskipws`.

Каждая из двух данных последовательностей считается составленной из пяти строк, разделенных пробелами, если для чтения используются операторы ввода с типами `const char*` или `string`:

```
A fine and private place
"A fine and private place"
```

Наличие кавычек не делает пробелы внутри закавыченной строки ее частью. Просто открывающая кавычка становится начальным символом первого слова, а закрывающая – конечным символом последнего.

Вместо того чтобы читать из стандартного ввода по одному символу, можно воспользоваться потоковым итератором `istream_iterator`:

```
#include <algorithm>
#include <string>
#include <vector>
#include <iostream>

int main()
{
          istream_iterator< string > in( cin ), eos ;
          vector< string > text ;

          // копировать прочитанные из стандартного ввода значения
     // в вектор text
          copy( in , eos , back_inserter( text ) ) ;

          sort( text.begin() , text.end() ) ;

          // удалить дубликаты
          vector< string >::iterator it;
          it = unique( text.begin() , text.end() ) ;
          text.erase( it , text.end() ) ;

          // вывести получившийся вектор
          int line_cnt = 1 ;
          for ( vector< string >::iterator iter = text.begin() ;
          iter != text.end() ; ++iter , ++line_cnt )
              cout << *iter
                   << ( line_cnt % 9 ? " " : "\n" ) ;

          cout << endl;
}
```

Пусть входом для этой программы будет файл `istream_iter.C` с исходным текстом. В системе UNIX мы можем перенаправить стандартный ввод на файл следующим образом (`istream_iter` – имя исполняемого файла программы):

```
istream_iter < istream_iter.C
```

(Для других систем необходимо изучить документацию.) В результате программа выводит:

```
!= " " "\n" #include % ( ) *iter ++iter
++line_cnt , 1 9 : ; << <algorithm> <iostream.h>
<string> <vector> = > >::difference_type >::iterator ? allocator
back_inserter(
cin copy( cout diff_type eos for in in( int
istream_iterator< it iter line_cnt main() sort( string test test.begin()
test.end() test.erase( typedef unique( vector< { }
```

(Потоковые итераторы ввода/вывода `iostream` рассматривались в разделе 12.4.)

Помимо предопределенных операторов ввода, можно определить и собственные перегруженные экземпляры для считывания в пользовательские типы данных. (Подробнее мы расскажем об этом в разделе 20.5.)

20.2.1. Строковый ввод

Считывание можно производить как в С-строки, так и в объекты класса `string`. Мы рекомендуем пользоваться последними. Их главное преимущество – автоматическое управление памятью для хранения символов. Чтобы прочитать данные в С-строку, т.е. массив символов, необходимо сначала задать его размер, достаточный для хранения строки. Обычно мы читаем символы в буфер, затем выделяем из хипа ровно столько памяти, сколько нужно для хранения прочитанной строки, и копируем данные из буфера

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

char inBuf[ 1024 ];
try
{
   while ( cin >> inBuf ) {
           char *str = new char[ strlen( inBuf ) + 1 ];
           strcpy( str, inBuf );
           // ... сделать что-то с массивом символов str
           delete [] str;
   }
}
```

в эту память:

```
catch( ... ) { delete [] str; throw; }
```

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

string str;
while ( cin >> str )
```

Работать с типом `string` значительно проще:

```
        // ... сделать что-то со строкой
```

Рассмотрим операторы ввода в С-строки и в объекты класса `string`. В качестве входного текста по-прежнему будет использоваться рассказ об Алисе Эмме:

```
Alice Emma has long flowing red hair. Her Daddy says
when the wind blows through her hair, it looks almost
alive, like a fiery bird in flight. A beautiful fiery
bird, he tells her, magical but untamed. "Daddy, shush,
there is no such creature," she tells him, at the same time
wanting him to tell her more. Shyly, she asks, "I mean,
Daddy, is there?"
```

Поместим этот текст в файл `alice_emma`, а затем перенаправим на него стандартный вход программы. Позже, когда мы познакомимся с файловым вводом, мы откроем и прочтем этот файл непосредственно. Следующая программа помещает прочитанные со стандартного ввода слова в С-строку и находит самое длинное слово:

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

int main()
{
          const int bufSize = 24;
          char buf[ bufSize ], largest[ bufSize ];

          // для хранения статистики
          int curLen, max = -1, cnt = 0;
          while ( cin >> buf )
          {
          curLen = strlen( buf );
          ++cnt;

          // новое самое длинное слово? сохраним его
          if ( curLen > max ) {
              max = curLen;
              strcpy( largest, buf );
          }
          }

          cout << "Число прочитанных слов "
               << cnt << endl;

     cout << "Длина самого длинного слова "
               << max << endl;

          cout << "Самое длинное слово "
               << largest << endl;

}
```

После компиляции и запуска программа выводит следующие сведения:

```
Число прочитанных слов 65
Длина самого длинного слова 10
Самое длинное слово creature,"
```

На самом деле этот результат неправилен: самое длинное слово `beautiful`, в нем девять букв. Однако выбрано `creature`, потому что программа сочла его частью запятую и кавычку. Следовательно, необходимо отфильтровать небуквенные символы.

Но прежде чем заняться этим, рассмотрим программу внимательнее. В ней каждое слово помещается в массив `buf`, длина которого равна `24`. Если бы в тексте попалось слово длиной `24` символа (или более), то буфер переполнился бы и программа, вероятно, закончилась бы крахом. Чтобы предотвратить переполнение входного массива, можно воспользоваться манипулятором `setw()`. Модифицируем предыдущую программу:

```
while ( cin >> setw( bufSize ) >> buf )
```

Здесь `bufSize` – размер массива символов `buf`. `setw()` разбивает строку длиной `bufSize` или больше на несколько строк, каждая из которых не длиннее, чем `bufSize - 1`.

Завершается такая частичная строка двоичным нулем. Для использования `setw()` в программу необходимо включить заголовочный файл `iomanip`:

```
#include <iomanip>
```

Если в объявлении массива buf размер явно не указан:

```
char buf[] = "Нереалистичный пример";
```

то программист может применить оператор sizeof, но при условии, что идентификатор является именем массива и находится в области видимости выражения:

```
while ( cin >> setw(sizeof( buf )) >> buf )
```

```
#include <iostream>
#include <iomanip>

int main()
{
          const int bufSize = 24;
          char buf[ bufSize ];
          char *pbuf = buf;

          // если строка длиннее, чем sizeof(char*),
          // она разбивается на несколько строк

          while ( cin >> setw( sizeof( pbuf )) >> pbuf )
             cout << pbuf << endl;
}
```

Применение оператора sizeof в следующем примере дает неожиданный результат:

Программа печатает:

```
$ a.out
The winter of our discontent

The
win
ter
of
our
dis
con
ten
t
```

Функции setw() вместо размера массива передается размер указателя, длина которого на нашей машине равна четырем байтам, поэтому вывод разбит на строки по три символа.

Попытка исправить ошибку приводит к еще более серьезной проблеме:

```
while ( cin >> setw(sizeof( *pbuf )) >> pbuf )
```

Мы хотели передать setw() размер массива, адресуемого pbuf. Но выражение

```
*pbuf
```

дает только один символ, т.е. объект типа char. Поэтому setw() передается значение 1. На каждой итерации цикла while в массив, на который указывает pbuf, помещается

только нулевой символ. До чтения из стандартного ввода дело так и не доходит, программа зацикливается.

При использовании класса string все проблемы управления памятью исчезают, об этом

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

int main()
{
          string buf, largest;

          // для хранения статистики
          int curLen,   // длина текущего слова
              max = -1, // максимальная длина слова
              cnt = 0;  // счетчик прочитанных слов

          while ( cin >> buf )
          {
          curLen = buf.size();
          ++cnt;

          // новое самое длинное слово? сохраним его
          if ( curLen > max )
          {
              max = curLen;
              largest = buf;
          }
          }

          cout << "Число прочитанных слов " << cnt << endl;
     cout << "Длина самого длинного слова " << max << endl;
          cout << "Самое длинное слово " << largest << endl;
}
```

заботится сам string. Вот как выглядит наша программа в данном случае:

Однако запятая и кавычка по-прежнему считаются частью слова. Напишем функцию для

```
#include <string>
void filter_string( string &str )
{
     // элементы, подлежащие фильтрации
     string filt_elems( "\",?." );
     string::size_type pos = 0;
     while (( pos = str.find_first_of( filt_elems, pos ))
                 != string::npos )
                 str.erase( pos, 1 );
}
```

удаления этих символов из слова:

Эта функция работает правильно, но множество символов, которые мы собираемся отбрасывать, “зашито” в код. Лучше дать пользователю возможность самому передать строку, содержащую такие символы. Если он согласен на множество по умолчанию, то может передать пустую строку.

```
#include <string>
void filter_string( string &str,
                    string filt_elems = string("\",."))
{
     string::size_type pos = 0;
     while (( pos = str.find_first_of( filt_elems, pos ))
                 != string::npos )
                 str.erase( pos, 1 );
}
```

Более общая версия `filter_string()` принимает пару итераторов, обозначающих

```
template <class InputIterator>
void filter_string( InputIterator first, InputIterator last,
                    string filt_elems = string("\",."))
{
          for ( ; first != last; first++ )
          {
          string::size_type pos = 0;
          while (( pos = (*first).find_first_of( filt_elems, pos ))
                      != string::npos )
                      (*first).erase( pos, 1 );
          }
}
```

диапазон, где производится фильтрация:

С использованием этой функции программа будет выглядеть так:

```
#include <string>
#include <algorithm>
#include <iterator>
#include <vector>
#include <iostream>

bool length_less( string s1, string s2 )
          { return s1.size() < s2.size(); }

int main()
{
     istream_iterator< string > input( cin ), eos;

          vector< string > text;
     // copy - это обобщенный алгоритм
          copy( input, eos, back_inserter( text ));

          string filt_elems( "\",.;:");
          filter_string( text.begin(), text.end(), filt_elems );

          int cnt = text.size();
     // max_element - это обобщенный алгоритм
          string *max = max_element( text.begin(), text.end(),
                                     length_less );
          int len = max->size();

          cout << "Число прочитанных слов "
               << cnt << endl;

     cout << "Длина самого длинного слова "
               << len << endl;

          cout << "Самое длинное слово "
               << *max << endl;

}
```

Когда мы применили в алгоритме `max_element()` стандартный оператор “меньше”, определенный в классе `string`, то были удивлены полученным результатом:

```
Число прочитанных слов 65
Длина самого длинного слова 4
Самое длинное слово wind
```

Очевидно, что `wind` – это не самое длинное слово. Оказывается, оператор “меньше” в классе `string` сравнивает строки не по длине, а в лексикографическом порядке. И в этом смысле `wind` – действительно максимальный элемент. Для того чтобы найти слово максимальной длины, мы должны заменить оператор “меньше” предикатом `length_less()`. Тогда результат будет таким:

```
Число прочитанных слов 65
Длина самого длинного слова 9
Самое длинное слово beautiful
```

Упражнение 20.2

Прочитайте из стандартного ввода последовательность данных таких типов: `string`, `double`, `string`, `int`, `string`. Каждый раз проверяйте, не было ли ошибки чтения.

Упражнение 20.3

Прочитайте из стандартного ввода заранее неизвестное число строк. Поместите их в список. Найдите самую длинную и самую короткую строку.

20.3. Дополнительные операторы ввода/вывода

Иногда необходимо прочитать из входного потока последовательность не интерпретируемых байтов, а типов данных, таких, как `char`, `int`, `string` и т.д. Функция-член `get()` класса `istream` читает по одному байту, а функция `getline()` читает строку, завершающуюся либо символом перехода на новую строку, либо каким-то иным символом, определяемым пользователем. У функции-члена `get()` есть три формы:

- `get(char& ch)` читает из входного потока один символ (в том числе и пустой) и помещает его в `ch`. Она возвращает объект `iostream`, для которого была вызвана. Например, следующая программа собирает статистику о входном потоке, а затем

```
#include <iostream>

int main()
{
   char ch;
   int  tab_cnt = 0, nl_cnt = 0, space_cnt = 0,
        period_cnt = 0, comma_cnt = 0;
   while ( cin.get(ch)) {
           switch( ch ) {
                   case ' ':  space_cnt++;  break;
                   case '\t': tab_cnt++;    break;
                   case '\n': nl_cnt++;     break;
                   case '.':  period_cnt++; break;
                   case ',':  comma_cnt++;  break;
           }
           cout.put(ch);
   }

   cout << "\nнаша статистика:\n\t"
        << "пробелов: "    << space_cnt  << '\t'
        << "символов новой строки: " << nl_cnt     << '\t'
        << "табуляций: "      << tab_cnt    << "\n\t"
        << "точек: "   << period_cnt << '\t'
        << "запятых: "    << comma_cnt  << endl;
```

  копирует входной поток в выходной:

```
}
```

  Функция-член `put()` класса `ostream` дает альтернативный метод вывода символа в выходной поток: `put()` принимает аргумент типа `char` и возвращает объект класса `ostream`, для которого была вызвана.

После компиляции и запуска программа печатает следующий результат:

```
Alice Emma has long flowing red hair. Her Daddy says
when the wind blows through her hair, it looks almost alive,
like a fiery bird in flight. A beautiful fiery bird, he tells her,
```

```
magical but untamed. "Daddy, shush, there is no such creature,"
she tells him, at the same time wanting him to tell her more.
Shyly, she asks, "I mean, Daddy, is there?"

наша статистика:
       пробелов: 59      символов новой строки: 6     табуляций: 0
       точек: 4          запятых: 12
```

- вторая форма `get()` также читает из входного потока по одному символу, но возвращает не поток `istream`, а значение прочитанного символа. Тип возвращаемого значения равен `int`, а не `char`, поскольку необходимо возвращать еще и признак конца файла, который обычно равен `-1`, чтобы отличаться от кодов реальных символов. Для проверки на конец файла мы сравниваем полученное значение с константой `EOF`, определенной в заголовочном файле `iostream`. Переменная, в которой сохраняется значение, возвращенное `get()`, должна быть объявлена как `int`,

```
#include <iostream>

int main()
{
          int ch;

          // альтернатива:
     // while ( ch = cin.get() && ch != EOF )
          while (( ch = cin.get()) != EOF )
              cout.put( ch );

          return 0;
}
```

чтобы в ней можно было представить не только код любого символа, но и `EOF`:

При использовании любой из этих форм `get()` для чтения данной последовательности нужно семь итераций:

```
a b c
d
```

Читаются следующие символы: (`'a'`, пробел, `'b'`, пробел, `'c'`, символ новой строки, `'d'`). На восьмой итерации читается `EOF`. Оператор ввода (`>>`) по умолчанию пропускает пустые символы, поэтому на ту же последовательность потребуется четыре итерации, на которых возвращаются символы: `'a'`, `'b'`, `'c'`, `'d'`. А вот следующая форма `get()` может прочесть всю последовательность всего за две итерации;

- сигнатура третьей формы `get()` такова:

```
get(char *sink, streamsize size, char delimiter='\n')
```

`sink` – это массив, в который помещаются символы. `size` – это максимальное число символов, читаемых из потока `istream`. `delimiter` – это символ-ограничитель, при обнаружении которого чтение прекращается. Сам ограничитель не читается, а оставляется в потоке и будет прочитан следующим. Программисты часто забывают удалить его из потока перед вторым обращением к `get()`. Чтобы избежать этой

ошибки, в показанной ниже программе мы воспользовались функцией-членом `ignore()` класса `istream`. По умолчанию ограничителем является символ новой строки.

Символы читаются из потока, пока одно из следующих условий не окажется истинным. Как только это случится, в очередную позицию массива помещается двоичный нуль.

- прочитано `size-1` символов;
- встретился конец файла;
- встретился символ-ограничитель (еще раз напомним, что он остается в потоке и будет считан следующим).

Эта форма `get()` возвращает объект `istream`, для которого была вызвана (функция-член `gcount()` позволяет узнать количество прочитанных символов). Вот простой пример ее применения:

```
#include <iostream>

int main()
{
          const int max_line = 1024;
          char line[ max_line ];

          while ( cin.get( line, max_line ))
          {
          // читается не больше max_line - 1 символов,
          // чтобы оставить место для нуля
          int get_count = cin.gcount();
          cout << "фактически прочитано символов: "
               << get_count << endl;

          // что-то сделать со строкой

          // если встретился символ новой строки,
          // удалить его, прежде чем приступать к чтению следующей
          if ( get_count < max_line-1 )
              cin.ignore();
          }
}
```

Если на вход этой программы подать текст о юной Алисе Эмме, то результат будет выглядеть так:

```
фактически прочитано символов: 52
фактически прочитано символов: 60
фактически прочитано символов: 66
фактически прочитано символов: 63
фактически прочитано символов: 61
фактически прочитано символов: 43
```

Чтобы еще раз протестировать поведение программы, мы создали строку, содержащую больше `max_line` символов, и поместили ее в начало текста. Получили:

```
фактически прочитано символов: 1023
```

```
фактически прочитано символов: 528
фактически прочитано символов: 52
фактически прочитано символов: 60
фактически прочитано символов: 66
фактически прочитано символов: 63
фактически прочитано символов: 61
фактически прочитано символов: 43
```

По умолчанию `ignore()` читает и удаляет один символ из потока, для которого вызвана, но можно и явно задать ограничитель и количество пропускаемых символов. В общем виде ее сигнатура такова:

```
ignore( streamsize length = 1, int delim = traits::eof )
```

`ignore()` читает и отбрасывает `length` символов из потока или все символы до ограничителя включительно или до конца файла и возвращает объект `istream`, для которого вызвана.

Мы рекомендуем пользоваться функцией `getline()`, а не `get()`, поскольку она автоматически удаляет ограничитель из потока. Сигнатура `getline()` такая же, как у `get()` с тремя аргументами (и возвращает она тоже объект `istream`, для которого вызвана):

```
getline(char *sink, streamsize size, char delimiter='\n')
```

Поскольку и `getline()`, и `get()` с тремя аргументами могут читать `size` символов или меньше, то часто нужно "спросить" у объекта `istream`, сколько символов было фактически прочитано. Это позволяет сделать функция-член `gcount()`: она возвращает число символов, прочитанных при последнем обращении к `get()` или `getline()`.

Функция-член `write()` класса `ostream` дает альтернативный метод вывода массива символов. Вместо того чтобы выводить символы до завершающего нуля, она выводит указанное число символов, включая и внутренние нули, если таковые имеются. Вот ее сигнатура:

```
write( const char *sink, streamsize length )
```

Здесь `length` определяет, сколько символов выводить. `write()` возвращает объект класса `ostream`, для которого она вызвана.

Парной для функции `write()` из класса `ostream` является функция `read()` из класса `istream` с такой сигнатурой:

```
read( char* addr, streamsize size )
```

`read()` читает `size` соседних байт из входного потока и помещает их, начиная с адреса `addr`. Функция `gcount()` возвращает число байт, прочитанных при последнем обращении к `read()`. В свою очередь `read()` возвращает объект класса `istream`, для которого она вызвана. Вот пример использования `getline()`, `gcount()` и `write()`:

```
#include <iostream>

int main()
{
          const int lineSize = 1024;
          int lcnt = 0; // сколько строк прочитано
          int max = -1; // длина самой длинной строки

          char inBuf[ lineSize ];

          // читается до конца строки, но не более 1024 символов
          while (cin.getline( inBuf, lineSize ))
          {
          // сколько символов фактически прочитано
          int readin = cin.gcount();

          // статистика: счетчик строк, самая длинная строка
          ++lcnt;
          if ( readin > max )
              max = readin;

          cout << "Строка #" << lcnt
               << "\tПрочитано символов: " << readin << endl;

          cout.write( inBuf, readin).put('\n').put('\n');
          }

          cout << "Всего прочитано строк: " << lcnt << endl;
          cout << "Самая длинная строка: " << max << endl;
}
```

Когда на вход было подано несколько фраз из романа Германа Мелвилла “Моби Дик”, программа напечатала следующее:

```
Строка #1 Прочитано символов: 45
Call me Ishmael.  Some years ago, never mind

Строка #2 Прочитано символов: 46
how long precisely, having little or no money

Строка #3 Прочитано символов: 48
in my purse, and nothing particular to interest

Строка #4 Прочитано символов: 51
me on shore, I thought I would sail about a little

Строка #5 Прочитано символов: 47
and see the watery part of the world.  It is a

Строка #6 Прочитано символов: 43
way I have of driving off the spleen, and

Строка #7 Прочитано символов: 28
regulating the circulation.

Всего прочитано строк: 7
Самая длинная строка: 51
```

Функция-член `getline()` класса `istream` поддерживает только ввод в массив символов. Однако в стандартной библиотеке есть обычная функция `getline()`, которая помещает символы в объект класса `string`:

```
getline( istream &is, string str, char delimiter );
```

Эта функция читает не более str::max_size()-1 символов. Если входная последовательность длиннее, то операция завершается неудачно и объект переводится в ошибочное состояние. В противном случае ввод прекращается, когда прочитан ограничитель (он удаляется из потока, но в строку не помещается) либо достигнут конец файла.

```
// возвращает символ в поток
putback( char class );

// устанавливает "указатель на следующий символ потока istream на один
          символ назад
unget();

// возвращает следующий символ (или EOF),
// но не извлекает его из потока
```

Вот еще три необходимые нам функции-члена класса istream:

```
peek();
```

```
char ch, next, lookahead;

while ( cin.get( ch ))
{
   switch (ch) {
   case '/':
      // это комментарий? посмотрим с помощью peek()
      // если да, пропустить остаток строки
      next = cin.peek();
      if ( next == '/' )
         cin.ignore( lineSize, '\n' );
      break;
   case '>':
      // проверка на лексему >>=
      next = cin.peek();
      if ( next == '>' ) {
         lookahead = cin.get();
         next = cin.peek();
         if ( next != '=' )
            cin.putback( lookahead );
         }
         // ...
```

Следующий фрагмент иллюстрирует использование некоторых из них:

```
}
```

Упражнение 20.4

Прочитайте из стандартного ввода следующую последовательность символов, включая все пустые, и скопируйте каждый символ на стандартный вывод (эхо-копирование):

```
a  b c
d     e
f
```

Упражнение 20.5

Прочитайте фразу “riverrun, from bend of bay to swerve of shore” сначала как последовательность из девяти строк, а затем как одну строку.

Упражнение 20.6

С помощью функций `getline()` и `gcount()` прочитайте последовательность строк из стандартного ввода и найдите самую длинную (не забудьте, что строку, прочитанную за несколько обращений к `getline()`, нужно считать одной).

20.4. Перегрузка оператора вывода

Если мы хотим, чтобы наш тип класса поддерживал операции ввода/вывода, то необходимо перегрузить оба соответствующих оператора. В этом разделе мы рассмотрим, как перегружается оператор вывода. (Перегрузка оператора ввода – тема следующего раздела.) Например, для класса `WordCount` он выглядит так:

```
class WordCount {
    friend ostream&
           operator<<( ostream&, const WordCount& );

public:
    WordCount( string word, int cnt=1 );
    // ...
private:
    string word;
    int occurs;
};

ostream&
operator <<( ostream& os, const WordCount& wd )
{    // формат: <счетчик> слово
     os << "< " << " > " > "
        << wd.word;
     return os;
}
```

Проектировщик должен решить, следует ли выводить завершающий символ новой строки. Лучше этого не делать: поскольку операторы вывода для встроенных типов такой символ не печатают, пользователь ожидает аналогичного поведения и от операторов в других классах. Определенный нами в классе `WordCount` оператор вывода можно использовать вместе с любыми другими операторами:

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

int main()
{
   WordCount wd( "sadness", 12 );
   cout << "wd:\n" << wd << endl;
   return 0;
}
```

Программа печатает на терминале строки:

```
wd:
<12> sadness
```

Оператор вывода – это бинарный оператор, который возвращает ссылку на объект класса ostream. В общем случае структура определения перегруженного оператора вывода

```
// структура перегруженного оператора вывода
ostream&
operator <<( ostream& os, const ClassType &object )
{
   // произвольный код для подготовки объекта

   // фактическое число членов
   os << // ...

   // возвращается объект ostream
   return os;
}
```

выглядит так:

Первый его аргумент – это ссылка на объект ostream, а второй – ссылка (обычно константная) на объект некоторого класса. Возвращается ссылка на ostream. Значением всегда является объект ostream, для которого оператор вызывался.

Поскольку первым аргументом является ссылка, оператор вывода должен быть определен как обычная функция, а не член класса. (Объяснение см. в разделе 15.1.) Если оператору необходим доступ к неоткрытым членам, то следует объявить его другом класса. (О друзьях говорилось в разделе 15.2.)

Пусть Location – это класс, в котором хранятся номера строки и колонки вхождения

```
#include <iostream>

class Location {
    friend ostream& operator<<( ostream&, const Location& );
private:
    short _line;
    short _col;
};
ostream& operator <<( ostream& os, const Location& lc )
{
    // объект Loc выводится в виде:  < 10,37 >
    os << "<" << lc._line
       << "," << lc._col << "> ";

    return os;
}
```

слова. Вот его определение:

Изменим определение класса WordCount, включив в него вектор occurList объектов Location и объект word класса string:

```
#include <vector>
#include <string>
#include <iostream>
#include "Location.h"

class WordCount {
    friend ostream& operator<<( ostream&, const WordCount& );

public:
        WordCount() {}
        WordCount( const string &word ) : _word( word ) {}
        WordCount( const string &word, int ln, int col )
                : _word( word ){ insert_location( ln, col ); }

        string word()   const { return _word; }
        int    occurs() const { return _occurList.size(); }
        void   found( int ln, int col )
                    { insert_location( ln, col ); }

private:
        void insert_location( int ln, int col )
                { _occurList.push_back( Location( ln, col )); }

        string             _word;
        vector< Location > _occurList;

};
```

В классах `string` и `Location` определен оператор вывода `operator<<()`. Так выглядит

```
ostream&
operator <<( ostream& os, const WordCount& wd )
{
    os << "<" << wd._occurList.size() << "> "
       << wd._word << endl;

    int cnt = 0, onLine = 6;
    vector< Location >::const_iterator first =
            wd._occurList.begin();
    vector< Location >::const_iterator last  =
            wd._occurList.end();

    for ( ; first != last; ++first )
    {
        // os << Location
        os << *first << " ";

       // форматирование: по 6 в строке
       if ( ++cnt >= onLine )
          { os << "\n"; cnt = 0;  }

    }
    return os;
}
```

измененное определение оператора вывода в `WordCount`:

А вот небольшая программа для тестирования нового определения класса `WordCount`; позиции вхождений для простоты “зашиты” в код:

```
int main()
{
    WordCount search( "rosebud" );

    // для простоты явно введем 8 вхождений
    search.found(11,3);  search.found(11,8);
    search.found(14,2);  search.found(34,6);
    search.found(49,7);  search.found(67,5);
    search.found(81,2);  search.found(82,3);
    search.found(91,4);  search.found(97,8);

    cout << "Вхождения: " << "\n"
         << search << endl;

    return 0;
}
```

После компиляции и запуска программа выводит следующее:

```
Вхождения:
<10> rosebud
<11,3>  <11,8>  <14,2>  <34,6>  <49,7>  <67,5>
<81,2>  <82,3>  <91,4>  <97,8>
```

Полученный результат сохранен в файле `output`. Далее мы определим оператор ввода, с помощью которого прочитаем данные из этого файла.

Упражнение 20.7

```
class Date {
public:
   // ...
private:
   int month, day, year;
};
```

Дано определение класса `Date`:

Напишите перегруженный оператор вывода даты в формате:

(a)

```
// полное название месяца
September 8th, 1997
```

(b)

```
9 / 8 / 97
```

(c) Какой формат лучше? Объясните.

(d) Должен ли оператор вывода `Date` быть функцией-другом? Почему?

Упражнение 20.8

Определите оператор вывода для следующего класса `CheckoutRecord`:

```
class CheckoutRecord {       // запись о выдаче
public:
   // ...
private:
   double book_id;          // идентификатор книги
   string title;            // название
   Date date_borrowed;      // дата выдачи
   Date date_due;           // дата возврата
   pair<string,string> borrower;  // кому выдана
   vector pair<string,string> wait_list;   // очередь на книгу
};
```

20.5. Перегрузка оператора ввода

Перегрузка оператора ввода (>>) похожа на перегрузку оператора вывода, но, к сожалению, возможностей для ошибок гораздо больше. Вот, например, его реализация для класса WordCount:

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

/* необходимо модифицировать определение класса WordCount, чтобы
   оператор ввода был другом
   class WordCount {
      friend ostream& operator<<( ostream&, const WordCount& );
      friend istream& operator>>( istream&, const WordCount& );
*/

istream&
operator >>( istream &is, WordCount &wd )
{
    /* формат хранения объекта WordCount:
     * <2> строка
     * <7,3> <12,36>
     */

    int ch;

    /* прочитать знак '<'. Если его нет,
     * перевести поток в ошибочное состояние и выйти
     */
    if ((ch = is.get()) != '<' )
    {
        // is.setstate( ios_base::badbit );
        return is;
    }

    // прочитать длину
    int occurs;
    is >> occurs;

    // читать до обнаружения >; ошибки не контролируются
    while ( is && (ch = is.get()) != '>' ) ;

    is >> wd._word;

    // прочитать позиции вхождений;
    // каждая позиция имеет формат: < строка, колонка >
    for ( int ix = 0; ix < occurs; ++ix )
    {
        int line, col;
        // извлечь значения
        while (is && (ch = is.get())!= '<' ) ;
        is >> line;

        while (is && (ch = is.get())!= ',' ) ;
        is >> col;

        while (is && (ch = is.get())!= '>' ) ;

        wd._occurList.push_back( Location( line, col ));
    }
    return is;
}
```

На этом примере показан целый ряд проблем, имеющих отношение к возможным ошибочным состояниям входного потока:

- поток, чтение из которого невозможно из-за неправильного формата, переводится в состояние `fail`:

```
is.setstate( ios_base::failbit );
```

- операции вставки и извлечения из потока, находящегося в ошибочном состоянии, не работают:

```
while (( ch = is.get() ) != lbrace)
```

Инструкция зациклится, если объект `istream` будет находиться в ошибочном состоянии. Поэтому перед каждым обращением к `get()` проверяется отсутствие

```
// проверить, находится ли поток "is" в "хорошем" состоянии
```

ошибки:

```
while ( is && ( ch = is.get() ) != lbrace)
```

Если объект `istream` не в “хорошем” состоянии, то его значение будет равно `false`. (О состояниях потока мы расскажем в разделе 20.7.)

Данная программа считывает объект класса `WordCount`, сохраненный оператором вывода

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

int main()
{
    WordCount readIn;

    // operator>>( cin, readIn )
    cin >> readIn;

    if ( !cin ) {
        cerr << "Ошибка ввода WordCount" << endl;
        return -1;
    }

    // operator<<( cout, readIn )
    cout << readIn << endl;
```

из предыдущего раздела:

```
}
```

Выводится следующее:

```
<10> rosebud
<11,3>  <11,8>  <14,2>  <34,6>  <49,7>  <67,5>
<81,2>  <82,3>  <91,4>  <97,8>
```

Упражнение 20.9

Оператор ввода класса `WordCount` сам читает объекты класса `Location`. Вынесите этот код в отдельный оператор ввода класса `Location`.

Упражнение 20.10

Реализуйте оператор ввода для класса `Date` из упражнения 20.7 в разделе 20.4.

Упражнение 20.11

Реализуйте оператор ввода для класса `CheckoutRecord` из упражнения 20.8 в разделе 20.4.

20.6. Файловый ввод/вывод

Если программе необходимо работать с файлом, то следует включить в нее заголовочный файл `fstream` (который в свою очередь включает `iostream`):

```
#include <fstream>
```

Если файл будет использоваться только для вывода, мы определяем объект класса `ofstream`. Например:

```
ofstream outfile( "copy.out", ios::base::out );
```

Передаваемые конструктору аргументы задают имя открываемого файла и режим открытия. Файл типа `ofstream` может быть открыт либо – по умолчанию – в режиме вывода (`ios_base::out`), либо в режиме дозаписи (`ios_base::app`). Такое определение

```
// по умолчанию открывается в режиме вывода
```

файла `outfile2` эквивалентно приведенному выше:

```
ofstream outfile2( "copy.out" );
```

Если в режиме вывода открывается существующий файл, то все хранившиеся в нем данные пропадают. Если же мы хотим не заменить, а добавить данные, то следует открывать файл в режиме дозаписи: тогда новые данные помещаются в конец. Если указанный файл не существует, то он создается в любом режиме.

Прежде чем пытаться прочитать из файла или записать в него, нужно проверить, что

```
if ( ! outfile ) { // открыть файл не удалось
   cerr << "не могу открыть "copy.out" для записи\n";
   exit( -1 );
```

файл был успешно открыт:

```
}
```

Класс `ofstream` является производным от `ostream`. Все определенные в `ostream`

```
char ch = ' ';
outFile.put( '1' ).put( ')' ).put( ch );
```

операции применимы и к `ofstream`. Например, инструкции

```
outFile << "1 + 1 = " << (1 + 1) << endl;
```

выводят в файл `outFile` последовательность символов:

```
1) 1 + 1 = 2
```

Следующая программа читает из стандартного ввода символы и копирует их в

```
#include <fstream>

int main()
{
    // открыть файл copy.out для вывода
    ofstream outFile( "copy.out" );

    if ( ! outFile ) {
        cerr << "Не могу открыть 'copy.out' для вывода\n";
        return -1;
    }

    char ch;
    while ( cin.get( ch ) )
              outFile.put( ch );
```

стандартный вывод:

```
}
```

К объекту класса `ofstream` можно применять и определенные пользователем экземпляры оператора вывода. Данная программа вызывает оператор вывода класса `WordCount` из

```
#include <fstream>
#include "WordCount.h"

int main()
{
    // открыть файл word.out для вывода
    ofstream oFile( "word.out" );
    // здесь проверка успешности открытия ...

    // создать и вручную заполнить объект WordCount
    WordCount artist( "Renoir" );
    artist.found( 7, 12 ); artist.found( 34, 18 );

    // вызывается оператор <<(ostream&, const WordCount&);
    oFile << artist;
```

предыдущего раздела:

```
}
```

Чтобы открыть файл только для чтения, применяется объект класса `ifstream`, производного от `istream`. Следующая программа читает указанный пользователем файл и копирует его содержимое на стандартный вывод:

```
#include <fstream>
#include <string>

int main()
{
    cout << "filename: ";
    string file_name;

    cin >> file_name;

    // открыть файл для ввода
    ifstream inFile( file_name.c_str() );

    if ( !inFile ) {
         cerr << "не могу открыть входной файл: "
              << file_name << " -- аварийный останов!\n";
         return -1;
    }

    char ch;
    while ( inFile.get( ch ))
            cout.put( ch );
}
```

Программа, показанная ниже, читает наш текстовый файл `alice_emma`, фильтрует его с помощью функции `filter_string()` (см. раздел 20.2.1, где приведены текст этой функции и содержимое файла), сортирует строки, удаляет дубликаты и записывает результат на стандартный вывод:

```
#include <fstream>
#include <iterator>
#include <vector>
#include <algorithm>

template <class InputIterator>
void filter_string( InputIterator first, InputIterator last,
                    string filt_elems = string("\",?."))
{
    for ( ; first != last; first++ )
    {
       string::size_type pos = 0;
       while (( pos = (*first).find_first_of( filt_elems, pos ))
                    != string::npos )
             (*first).erase( pos, 1 );
    }
}

int main()
{
          ifstream infile( "alice_emma" );

          istream_iterator<string> ifile( infile );
          istream_iterator<string> eos;

          vector< string > text;
          copy( ifile, eos, inserter( text, text.begin() ));

          string filt_elems( "\",.?;:" );
          filter_string( text.begin(), text.end(), filt_elems );

          vector<string>::iterator iter;

          sort( text.begin(), text.end() );
          iter = unique( text.begin(), text.end() );
          text.erase( iter, text.end() );

          ofstream outfile( "alice_emma_sort" );

          iter = text.begin();
          for ( int line_cnt = 1; iter != text.end();
           ++iter, ++line_cnt )
          {
        outfile << *iter << " ";
        if ( ! ( line_cnt % 8 ))
           outfile << '\n';
          }
     outfile << endl;

}
```

После компиляции и запуска программа выводит следующее:

```
A Alice Daddy Emma Her I Shyly a
alive almost asks at beautiful bird blows but
creature fiery flight flowing hair has he her
him in is it like long looks magical
mean more no red same says she shush
such tell tells the there through time to
untamed wanting when wind
```

Объекты классов `ofstream` и `ifstream` разрешено определять и без указания имени файла. Позже к этому объекту можно присоединить файл с помощью функции-члена

```
ifstream curFile;
// ...
curFile.open( filename.c_str() );
if ( ! curFile )  // открытие успешно?
```

`open()`:

```
   // ...
```

```
#include <fstream>

const int fileCnt = 5;
string fileTabl[ fileCnt ] = {
    "Melville", "Joyce", "Musil", "Proust", "Kafka"
};

int main()
{
    ifstream inFile; // не связан ни с каким файлом

    for ( int ix = 0; ix < fileCnt; ++ix )
    {
        inFile.open( fileTabl[ix].c_str() );
        // ... проверить успешность открытия
        // ... обработать файл
        inFile.close();
    }
```

Чтобы закрыть файл (отключить от программы), вызываем функцию-член `close()`:

```
}
```

Объект класса `fstream` (производного от `iostream`) может открывать файл для ввода *или* вывода. В следующем примере файл `word.out` сначала считывается, а затем записывается с помощью объекта типа `fstream`. Созданный ранее в этом разделе файл `word.out` содержит объект `WordCount`:

```
#include <fstream>
#include "WordCount.h"

int main()
{
    WordCount wd;
    fstream file;

    file.open( "word.out", ios::in );
    file >> wd;
    file.close();

    cout << "Прочитано: " << wd << endl;

    // операция ios_base::out стерла бы текущие данные
    file.open( "word.out", ios::app );
    file << endl << wd << endl;
    file.close();
}
```

Объект класса `fstream` может также открывать файл одновременно для ввода *и* вывода. Например, приведенная инструкция открывает файл `word.out` для ввода и дозаписи:

```
fstream io( "word.out", ios_base::in|ios_base::app );
```

Для задания нескольких режимов используется оператор побитового ИЛИ. Объект класса `fstream` можно позиционировать с помощью функций-членов `seekg()` или `seekp()`. Здесь буква `g` обозначает позиционирование для *чтения* (getting) символов (используется с объектом класса `ofstream`), а `p` – для *записи* (putting) символов (используется с объектом класса `ifstream`). Эти функции делают текущим тот байт в файле, который

```
// установить абсолютное смещение в файле
seekg( pos_type current_position )

// смещение от текущей позиции в том или ином направлении
```

имеет указанное абсолютное или относительное смещение. У них есть два варианта:

```
seekg( off_type offset_position, ios_base::seekdir dir );
```

В первом варианте текущая позиция устанавливается в некоторое абсолютное значение, заданное аргументом `current_position`, причем значение `0` соответствует началу файла. Например, если файл содержит такую последовательность символов:

```
abc def ghi jkl
```

то вызов

```
io.seekg( 6 );
```

позиционирует `io` на шестой символ, т.е. на `f`. Второй вариант устанавливает указатель рабочей позиции файла на заданное расстояние от текущей, от начала файла или от его конца в зависимости от аргумента `dir`, который может принимать следующие значения:

- ios_base::beg – от начала файла;
- ios_base::cur – от текущей позиции;
- ios_base::end – от конца файла.

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

В следующем примере каждый вызов seekg() позиционирует файл на i-ую запись:

```
readFile.ssekg( i * sizeof(Record), ios_base::beg );
```

С помощью первого аргумента можно задавать отрицательное значение. Переместимся на 10 байтов назад от текущей позиции:

```
readFile.seekg( -10, ios_base::cur );
```

Текущая позиция чтения в файле типа fstream возвращается любой из двух функций-членов tellg() или tellp(). Здесь 'p' означает запись (putting) и используется с

```
// сохранить текущую позицию
ios_base::pos_type mark = writeFile.tellp();

// ...
if ( cancelEntry )
   // вернуться к сохраненной позиции
```

объектом ofstream, а 'g' говорит о чтении (getting) и обслуживает объект ifstream:

```
   writeFile.seekp( mark );
```

Если необходимо сместиться вперед от текущей позиции на одну запись типа Record, то

```
// эквивалентные вызовы seekg
readFile.seekg( readFile.tellg() + sizeof(Record) );

// данный вызов считается более эффективным
```

можно воспользоваться любой из данных инструкций:

```
readFile.seekg( sizeof(Record), ios_base::cur );
```

Разберем реальный пример. Дан текстовый файл, нужно вычислить его длину в байтах и сохранить ее в конце файла. Кроме того, каждый раз при встрече символа новой строки требуется сохранить текущее смещение в конце файла. Вот наш текстовый файл:

```
abcd
efg
hi
j
```

Программа должна создать файл, модифицированный следующим образом:

```
abcd
efg
hi
j
5 9 12 14 24
```

```
#include <iostream>
#include <fstream>

main() {
    // открыть файл для ввода и дозаписи
    fstream inOut( "copy.out", ios_base::in|ios_base::app );
    int cnt = 0;   // счетчик байтов
    char ch;

    while ( inOut.get( ch ))
    {
        cout.put( ch ); // скопировать на терминал
        ++cnt;
        if ( ch == '\n' ) {
            inOut << cnt ;
            inOut.put( ' ' ); // пробел
        }
    }

    // вывести окончательное значение счетчика байтов
    inOut << cnt << endl;
    cout << "[ " << cnt << " ]" << endl;
    return 0;
}
```

Так выглядит первая попытка реализации:

`inOut` – это объект класса `fstream`, связанный с файлом `copy.out`, открытым для ввода и дозаписи. Если файл открыт в режиме дозаписи, то все новые данные записываются в конец.

При чтении любого (включая пробельные) символа, кроме конца файла, мы увеличиваем переменную `cnt` на `1` и копируем прочитанный символ на терминал, чтобы вовремя заметить ошибки в работе программы.

Встретив символ новой строки, мы записываем текущее значение `cnt` в `inOut`. Как только будет достигнут конец файла, цикл прекращается. Окончательное значение `cnt` выводится в файл и на экран.

Программа компилируется без ошибок и кажется правильной. Но если подать на вход несколько фраз из романа “Моби Дик” Германа Мелвилла:

```
Call me Ishmael.  Some years ago, never mind
how long precisely, having little or no money
in my purse, and nothing particular to interest
me on shore, I thought I would sail about a little
and see the watery part of the world.  It is a
way I have of driving off the spleen, and
regulating the circulation.
```

то получим такой результат:

```
[ 0 ]
```

Программа не вывела ни одного символа, видимо, полагая, что файл пуст. Проблема в том, что файл открыт для дозаписи и потому позиционирован на конец. При выполнении инструкции

```
inOut.get( ch );
```

мы читаем конец файла, цикл `while` завершается и выводится значение `0`.

Хотя мы допустили серьезную ошибку, исправить ее совсем несложно, поскольку причина понятна. Надо лишь перед чтением переустановить файл на начало. Это делается с помощью обращения:

```
inOut.seekg( 0 );
```

Запустим программу заново. На этот раз она печатает:

```
Call me Ishmael.  Some years ago, never mind
[ 45 ]
```

Как видим, выводится лишь первая строка текста и счетчик для нее, а оставшиеся шесть строк проигнорированы. Ну что ж, исправление ошибок – неотъемлемая часть профессии программиста. А проблема опять в том, что файл открыт в режиме дозаписи. Как только мы в первый раз вывели `cnt`, файл оказался позиционирован на конец. При следующем обращении к `get()` читается конец файла, и цикл `while` снова завершается преждевременно.

Нам необходимо встать на ту позицию в файле, где мы были перед выводом `cnt`. Для

```
// запомнить текущую позицию
ios_base::pos_type mark = inOut.tellg();
inOut << cnt << sp;
```

этого понадобятся еще две инструкции:

```
inOut.seekg( mark );   // восстановить позицию
```

После повторной компиляции программа выводит на экран ожидаемый результат. Но посмотрев на выходной файл, мы обнаружим, что она все еще не вполне правильна: окончательное значение счетчика есть на экране, но не в файле. Оператор вывода, следующий за циклом `while`, не был выполнен.

Дело в том, что `inOut` находится в состоянии “конец файла”, в котором операции ввода и вывода *не* выполняются. Для решения проблемы необходимо сбросить это состояние с помощью функции-члена `clear()`:

```
inOut.clear();   // обнулить флаги состояния
```

Окончательный вариант программы выглядит так:

```
#include <iostream>
#include <fstream>

int main()
{
    fstream inOut( "copy.out", ios_base::in|ios_base::app );
    int cnt=0;
    char ch;

    inOut.seekg(0);

    while ( inOut.get( ch ))
    {
        cout.put( ch );
        cnt++;

        if ( ch == '\n' )
        {
            // запомнить текущую позицию
            ios_base::pos_type mark = inOut.tellg();
            inOut << cnt << ' ';
            inOut.seekg( mark ); // восстановить позицию
        }
    }
    inOut.clear();
    inOut << cnt << endl;

    cout << "[ " << cnt << " ]\n";

    return 0;
}
```

Вот теперь – наконец-то! – все правильно. При реализации этой программы было необходимо явно сформулировать поведение, которое мы собирались поддержать. А каждое наше исправление было реакцией на выявившуюся ошибку вместо анализа проблемы в целом.

Упражнение 20.12

Пользуясь операторами вывода для класса `Date`, которые вы определили в упражнении 20.7, или для класса `CheckoutRecord` из упражнения 20.8 (см. раздел 20.4), напишите программу, позволяющую создать файл и писать в него.

Упражнение 20.13

Напишите программу для открытия и чтения файла, созданного в упражнении 20.12. Выведите содержимое файла на стандартный вывод.

Упражнение 20.14

Напишите программу для открытия файла, созданного в упражнении 20.12, для чтения и дозаписи. Выведите экземпляр класса `Date` или `CheckoutRecord`:

(a) в начало файла

(b) после второго из существующих объектов

(c) в конец файла

20.7. Состояния потока

Пользователей библиотеки iostream, разумеется, интересует, находится ли поток в

```
int ival;
```

ошибочном состоянии. Например, если мы пишем

```
cin >> ival;
```

и вводим слово "Borges", то cin переводится в состояние ошибки после неудачной попытки присвоить строковый литерал целому числу. Если бы мы ввели число 1024, то чтение прошло бы успешно и поток остался бы в нормальном состоянии.

Чтобы выяснить, в каком состоянии находится поток, достаточно проверить его значение

```
if ( !cin )
```

на истину:

```
   // операция чтения не прошла или встретился конец файла
```

```
while ( cin >> word )
```

Для чтения заранее неизвестного количества элементов мы обычно пишем цикл while:

```
      // операция чтения завершилась успешно ...
```

Условие в цикле while будет равно false, если достигнут конец файла или произошла ошибка при чтении. В большинстве случаев такой проверки потокового объекта достаточно. Однако при реализации оператора ввода для класса WordCount из раздела 20.5 нам понадобился более точный анализ состояния.

У любого потока есть набор флагов, с помощью которых можно следить за состоянием потока. Имеются четыре предикатные функции-члена:

```
if ( inOut.eof() )
```

- eof() возвращает true, если достигнут конец файла:

```
   // отлично: все прочитано ...
```

- bad() возвращает true при попытке выполнения некорректной операции, например при установке позиции за концом файла. Обычно это свидетельствует о том, что поток находится в состоянии ошибки;
- fail() возвращает true, если операция завершилась неудачно, например не удалось открыть файл или передан некорректный формат ввода:

```
ifstream iFile( filename, ios_base::in );
if ( iFile.fail() )    // не удалось открыть
   error_message( ... );
```

- good() возвращает true, если все вышеперечисленные условия ложны:

```
if ( inOut.good() )
```

Существует два способа явно изменить состояние потока iostream. С помощью функции-члена clear() ему явно присваивается указанное значение. Функция setstate() не сбрасывает состояние, а устанавливает один из флагов, не меняя значения остальных. Например, в коде оператора ввода для класса WordCount при обнаружении неверного формата мы используем setstate() для установки флага fail в состоянии

```
if ((ch = is.get()) != '<' )
{
   is.setstate( ios_base::failbit );
   return is;
```

объекта istream:

```
}
```

```
ios_base::badbit
ios_base::eofbit
ios_base::failbit
```

Имеются следующие значения флагов состояния:

```
ios_base::goodbit
```

Для установки сразу нескольких флагов используется побитовый оператор ИЛИ:

```
is.setstate( ios_base::badbit | ios_base::failbit );
```

```
if ( !cin ) {
   cerr << "Ошибка ввода WordCount" << endl;
   return -1;
```

При тестировании оператора ввода в классе WordCount (см. раздел 20.5) мы писали:

```
}
```

Возможно, вместо этого мы предпочли бы продолжить выполнение программы, предупредив пользователя об ошибке и попросив повторить ввод. Но перед чтением нового значения из потока cin необходимо перевести его в нормальное состояние. Это можно сделать с помощью функции-члена clear():

```
cin.clear(); // сброс ошибок
```

В более общем случае `clear()` используется для сброса текущего состояния и установки одного или нескольких флагов нового. Например:

```
cin.clear( ios_base::goodbit );
```

восстанавливает нормальное состояние потока. (Оба вызова эквивалентны, поскольку `goodbit` является для `clear()` аргументом по умолчанию.)

```
ios_base::iostate old_state = cin.rdstate();

cin.clear();
process_input();

// перевести поток cin в прежнее состояние
```

Функция-член `rdstate()` позволяет получить текущее состояние объекта:

```
cin.clear( old_state );
```

Упражнение 20.15

Измените один (или оба) оператор ввода для класса `Date` из упражнения 20.7 и/или класса `CheckoutRecord` из упражнения 20.8 (см. раздел 20.4) так, чтобы они устанавливали состояние объекта `istream`. Модифицируйте программы, которыми вы пользовались для тестирования этих операторов, для проверки явно установленного состояния, вывода его на печать и сброса в нормальное. Протестируйте программы, подав на вход правильные и неправильные данные.

20.8. Строковые потоки

Библиотека `iostream` поддерживает операции над строковыми объектами в памяти. Класс `ostringstream` вставляет символы в строку, `istringstream` читает символы из строкового объекта, а `stringstream` может использоваться как для чтения, так и для записи. Чтобы работать со строковым потоком, в программу необходимо включить заголовочный файл

```
#include <sstream>
```

Например, следующая функция читает весь файл `alice_emma` в объект `buf` класса `ostringstream`. Размер `buf` увеличивается по мере необходимости, чтобы вместить все символы:

```
#include <string>
#include <fstream>
#include <sstream>

string read_file_into_string()
{
          ifstream ifile( "alice_emma" );
          ostringstream buf;

          char ch;
          while ( buf && ifile.get( ch ))
                  buf.put( ch );
          return buf.str();
}
```

Функция-член `str()` возвращает строку – объект класса `string`, ассоциированный со строковым потоком `ostringstream`. Этой строкой можно манипулировать так же, как и “обычным” объектом класса `string`. Например, в следующей программе `text` почленно

```
int main()
{
          string text = read_file_into_string();

          // запомнить позиции каждого символа новой строки
          vector< string::size_type > lines_of_text;
          string::size_type pos = 0;

          while ( pos != string::npos )
          {
          pos = text.find( '\n' pos );
           lines_of_text.push_back( pos );
          }

          // ...
}
```

инициализируется строкой, ассоциированной с `buf`:

Объект класса `ostringstream` можно использовать для автоматического форматирования составной строки, т.е. строки, составленной из данных разных типов. Так, следующий оператор вывода автоматически преобразует любой арифметический тип в соответствующее строковое представление, поэтому заботиться о выделении нужного количества памяти нет необходимости:

```
#include <iostream>
#include <sstream>

int main()
{
          int    ival  = 1024;     int   *pival = &ival;
          double  dval  = 3.14159; double *pdval = &dval;

          ostringstream format_message;

          // преобразование значений в строковое представление
          format_message << "ival: "  << ival
                         << " адрес ival: " << pival << 'n'
                         << "dval: " << dval
                         << " адрес dval: " << pdval << endl;

          string msg = format_message.str();
          cout << " размер строки сообщения: " << msg.size()
               << " сообщение: " << msg << endl;

}
```

Иногда лучше собрать все диагностические сообщения об ошибках, а не выводить их по мере возникновения. Это легко сделать с помощью перегруженного множества функций

```
string
format( string msg, int expected, int received )
{
   ostringstream message;
   message << msg << " ожидалось: " << expected
           << " принято: " << received << "\n";
   return message.str();
}

string format( string msg, vector<int> *values );
```

форматирования:

```
// ... и так далее
```

Приложение может сохранить такие строки для последующего отображения и даже рассортировать их по серьезности. Обобщить эту идею помогают классы `Notify` (извещение), `Log` (протокол) и `Error` (ошибка).

Поток `istringstream` читает из объекта класса `string`, с помощью которого был сконструирован. В частности, он применяется для преобразования строкового представления числа в его арифметическое значение:

```
#include <iostream>
#include <sstream>
#include <string>

int main()
{
          int    ival = 1024;    int *pival = &ival;
          double dval = 3.14159; double *pdval = &dval;

          // создает строку, в которой значения разделены пробелами
          ostringstream format_string;

          format_string << ival << " " << pival << " "
                   << dval << " " << pdval << endl;

          // извлекает сохраненные значения в коде ASCII
          // и помещает их в четыре разных объекта
          istringstream input_istring( format_string.str() );

          input_istring >> ival >> pival
                   >> dval >> pdval;

}
```

Упражнение 20.16

В языке Си форматирование выходного сообщения производится с помощью функций

```
int    ival = 1024;
double dval = 3.14159;
char   cval = 'a';
char  *sval = "the end";

printf( "ival: %d\tdval% %g\tcval: %c\tsval: %s",
```

семейства `printf()`. Например, следующий фрагмент

```
         ival, dval, cval, sval );
```

печатает:

```
ival: 1024   dval: 3.14159  cval: a   sval: the end
```

Первым аргументом `printf()` является форматная строка. Каждый символ % показывает, что вместо него должно быть подставлено значение аргумента, а следующий за ним символ определяет тип этого аргумента. Вот некоторые из поддерживаемых типов

```
%d        целое число
%g        число с плавающей точкой
%c        char
```

(полное описание см. в [KERNIGHAN88]):

```
%s        C-строка
```

Дополнительные аргументы `printf()` на позиционной основе сопоставляются со спецификаторами формата, начинающимися со знака %. Все остальные символы в форматной строке рассматриваются как литералы и выводятся буквально.

Основные недостатки семейства функций `printf()` таковы: во-первых, форматная строка не обобщается на определенные пользователем типы, и, во-вторых, если типы или число аргументов не соответствуют форматной строке, компилятор не заметит ошибки, а вывод будет отформатирован неверно. Однако у функций `printf()` есть и достоинство – компактность записи.

1. Получите так же отформатированный результат с помощью объекта класса `ostringstream`.
2. Сформулируйте достоинства и недостатки обоих подходов.

20.9. Состояние формата

Каждый объект класса из библиотеки `iostream` поддерживает *состояние формата*, которое управляет выполнением операций форматирования, например основание системы счисления для целых значений или точность для значений с плавающей точкой. Для модификации состояния формата объекта в распоряжении программиста имеется предопределенный набор манипуляторов.[1] Манипулятор применяется к потоковому объекту так же, как к данным. Однако вместо чтения или записи данных манипулятор модифицирует внутреннее состояние потока. Например, по умолчанию объект типа `bool`,

Примечание [O.A.6]: Нумерация сносок сбита.

```
#include <iostream.h>

int main()
{
    bool illustrate = true;

    cout << "объект illustrate типа bool установлен в true: "
         << illustrate << '\n';
}
```

имеющий значение `true` (а также литеральная константа `true`), выводится как целая `'1'`:

Чтобы поток `cout` выводил переменную `illustrate` в виде слова `true`, мы применяем манипулятор `boolalpha`:

1 Кроме того, программист может устанавливать и сбрасывать флаги состояния формата с помощью функций-членов `setf()` и `unsetf()`. Мы их рассматривать не будем; исчерпывающие ответы на вопросы, относящиеся к этой теме, можно получить в [STROUSTRUP97].

```
#include <iostream.h>

int main()
{
    bool illustrate = true;
    cout << "объект illustrate типа bool установлен в true: ";

    // изменяет состояние cout так, что булевские значения
    // печатаются в виде строк true и false
    cout << boolalpha;
    cout << illustrate << '\n';

}
```

Поскольку манипулятор возвращает потоковый объект, к которому он применялся, то допустимо прицеплять его к выводимым данным и другим манипуляторам. Вот как

```
#include <iostream.h>

int main()
{
    bool illustrate = true;
    cout << "объект illustrate типа bool: "
         << illustrate
         << "\nс использованием boolalpha: "
         << boolalpha << illustrate << '\n';

    // ...
}
```

можно перемежать данные и манипуляторы в нашей программе:

Вывод данных и манипуляторов вперемежку может сбить пользователя с толку. Применение манипулятора изменяет не только представление следующего за ним объекта, но и внутреннее состояние потока. В нашем примере все значения типа `bool` в оставшейся части программы также будут выводиться в виде строк.

Чтобы отменить сделанную модификацию потока `cout`, необходимо использовать

```
cout << boolalpha   // устанавливает внутреннее состояние cout
     << illustrate
```

манипулятор `noboolalpha`:

```
     << noboolalpha  // сбрасывает внутреннее состояние cout
```

Как мы покажем, для многих манипуляторов имеются парные.

По умолчанию значения арифметических типов читаются и записываются в десятичной системе счисления. Программист может изменить ее на восьмеричную или шестнадцатеричную, а затем вернуться к десятичной (это распространяется только на целые типы, но не на типы с плавающей точкой), пользуясь манипуляторами `hex`, `oct` и `dec`:

```
#include <iostream>
int main()
{
          int ival = 16;
          double dval = 16.0;

          cout << "ival: " << ival
          << " установлен oct: " << oct << ival << "\n";

          cout << "dval: " << dval
          << " установлен hex: " << hex << dval << "\n";

          cout << "ival: " << ival
          << " установлен dec: " << dec << ival << "\n";

}
```

Эта программа печатает следующее:

```
ival: 16 установлен oct: 20
dval: 16 установлен hex: 16
ival: 10 установлен dec: 16
```

Но, глядя на значение, мы не можем понять, в какой системе счисления оно записано. Например, 20 – это действительно 20 или восьмеричное представление 16? Манипулятор showbase выводит основание системы счисления вместе со значением с помощью следующих соглашений:

- `0x` в начале обозначает шестнадцатеричную систему (если мы хотим, чтобы вместо строчной буквы `'x'` печаталась заглавная, то можем применить манипулятор uppercase, а для отмены – манипулятор nouppercase);
- `0` в начале обозначает восьмеричную систему;
- отсутствие того и другого обозначает десятичную систему.

```
#include <iostream>
int main()
{
          int ival = 16;
          double dval = 16.0;

     cout << showbase;

          cout << "ival: " << ival
          << " установлен oct: " << oct << ival << "\n";

          cout << "dval: " << dval
          << " установлен hex: " << hex << dval << "\n";

          cout << "ival: " << ival
          << " установлен dec: " << dec << ival << "\n";

     cout << noshowbase;
```

Вот та же программа, но и с использованием showbase:

```
}
```

Результат:

```
ival: 16 установлен oct: 020
dval: 16 установлен hex: 16
ival: 0x10 установлен dec: 16
```

Манипулятор noshowbase восстанавливает состояние cout, при котором основание системы счисления не выводится.

По умолчанию значения с плавающей точкой выводятся с точностью 6. Эту величину можно модифицировать с помощью функции-члена precision(int) или манипулятора setprecision(); для использования последнего необходимо включить заголовочный

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

int main()
{
          cout << "Точность: "
          << cout.precision() << endl
          << sqrt(2.0) << endl;

     cout.precision(12);
          cout << "\nТочность: "
               << cout.precision() << endl
          << sqrt(2.0) << endl;

          cout << "\nТочность: "  << setprecision(3)
               << cout.precision() << endl
               << sqrt(2.0) << endl;

          return 0;
}
```

файл iomanip. precision() возвращает текущее значение точности. Например:

После компиляции и запуска программа печатает следующее:

```
Точность: 6
1.41421

Точность: 12
1.41421356237

Точность: 3
1.41
```

Манипуляторы, принимающие аргумент, такие, как setprecision() и setw(), требуют включения заголовочного файла iomanip:

```
#include <iomanip>
```

Кроме описанных аспектов, setprecision() имеет еще два: на целые значения он не оказывает никакого влияния; значения с плавающей точкой округляются, а не

обрезаются. Таким образом, при точности 4 значение 3.14159 печатается как 3.142, а при точности 3 – как 3.14.

По умолчанию десятичная точка не печатается, если дробная часть значения равна 0. Например:

```
cout << 10.00
```

выводит

```
10
```

```
cout << showpoint
     << 10.0
```

Чтобы точка выводилась, воспользуйтесь манипулятором showpoint:

```
     << noshowpoint << '\n';
```

Манипулятор noshowpoint восстанавливает поведение по умолчанию.

По умолчанию значения с плавающей точкой выводятся в нотации с фиксированной точкой. Для перехода на научную нотацию используется идентификатор scientific, а

```
cout << "научная: " << scientific
     << 10.0
     << "с фиксированной точкой: " << fixed
```

для возврата к прежней нотации – модификатор fixed:

```
     << 10.0 << '\n';
```

В результате печатается:

```
научная: 1.0e+01
с фиксированной точкой: 10
```

Если бы мы захотели вместо буквы 'e' выводить 'E', то следовало бы употребить манипулятор uppercase, а для возврата к 'e' – nouppercase. (Манипулятор uppercase не приводит к переводу букв в верхний регистр при печати.)

По умолчанию перегруженные операторы ввода пропускают пустые символы (пробелы, знаки табуляции, новой строки и возврата каретки). Если дана последовательность:

```
a bc
d
```

то цикл

```
char ch;
while ( cin >> ch )
     // ...
```

читает все буквы от 'a' до 'd' за четыре итерации, а пробельные разделители оператором ввода игнорируются. Манипулятор noskipws отменяет такой пропуск пробельных

```
char ch;
cin >> noskipws;
while ( cin >> ch )
     // ...
```

символов:

```
cin >> skipws;
```

Теперь цикл while будет выполняться семь раз. Чтобы восстановить поведение по умолчанию, к потоку cin применяется манипулятор skipws.

Когда мы пишем:

```
cout << "пожалуйста, введите значение: ";
```

то в буфере потока cout сохраняется литеральная строка. Есть ряд условий, при которых буфер сбрасывается (т.е. опустошается), – в нашем случае в стандартный вывод:

- буфер может заполниться. Тогда перед чтением следующего значения его необходимо сбросить;
- буфер можно сбросить явно с помощью любого из манипуляторов flush, ends

```
// сбрасывает буфер
cout << "hi!" << flush;
// вставляет нулевой символ, затем сбрасывает буфер
char ch[2]; ch[0] = 'a'; ch[1] = 'b';
cout << ch << ends;

// вставляет символ новой строки, затем сбрасывает буфер
```

или endl:

```
cout << "hi!" << endl;
```

- при установлении внутренней переменной состояния потока unitbuf буфер сбрасывается после каждой операции вывода;
- объект ostream может быть *связан* (tied) с объектом istream. Тогда буфер ostream сбрасывается каждый раз, когда istream читает из входного потока. cout всегда связан с cin:

```
cin.tie( &cout );
```

Инструкция

```
cin >> ival;
```

приводит к сбросу буфера cout.

В любой момент времени объект ostream разрешено связывать только с одним объектом istream. Чтобы разорвать существующую связь, мы передаем функции-члену tie()

```
istream is;
ostream new_os;

// ...

// tie() возвращает существующую связь
ostream *old_tie = is.tie();

is.tie( 0 );   // разорвать существующую связь
is.tie( &new_os );  // установить новую связь

// ...

is.tie( 0 );   // разорвать существующую связь
```

значение 0:

```
is.tie( old_tie );  // восстановить прежнюю связь
```

Мы можем управлять шириной поля, отведенного для печати числового или строкового

```
#include <iostream>
#include <iomanip>

int main()
{
          int ival = 16;
          double dval = 3.14159;

          cout << "ival: " << setw(12) << ival << '\n'
               << "dval: " << setw(12) << dval << '\n';
}
```

значения, с помощью манипулятора setw(). Например, программа

печатает:

```
ival:           16
dval:      3.14159
```

Второй модификатор setw() необходим потому, что, в отличие от других манипуляторов, setw() не изменяет состояние формата объекта ostream.

Чтобы выровнять значение по левой границе, мы применяем манипулятор left (соответственно манипулятор right восстанавливает выравнивание по правой границе). Если мы хотим получить такой результат:

```
    16
-    3
```

то пользуемся манипулятором `internal`, который выравнивает знак по левой границе, а значение – по правой, заполняя пустое пространство пробелами. Если же нужен другой символ, то можно применить манипулятор `setfill()`. Так

```
cout << setw(6) << setfill('%') << 100 << endl;
```

печатает:

```
%%%100
```

В табл. 20.1 приведен полный перечень предопределенных манипуляторов.

Таблица 20.1. Манипуляторы

Манипулятор	Назначение
boolalpha	Представлять `true` и `false` в виде строк
*noboolalpha	Представлять `true` и `false` как 1 и 0
Showbase	Печатать префикс, обозначающий систему счисления
*noshowbase	Не печатать префикс системы счисления
showpoint	Всегда печатать десятичную точку
*noshowpoint	Печатать десятичную точку только в том случае, если дробная часть ненулевая
showpos	Печатать + для неотрицательных чисел
*noshowpos	Не печатать + для неотрицательных чисел
Манипулятор	**Назначение**
*skipws	Пропускать пробельные символы в операторах ввода
noskipws	Не пропускать пробельные символы в операторах ввода
uppercase	Печатать `0X` при выводе в шестнадцатеричной системе счисления; `E` – при выводе в научной нотации
*nouppercase	Печатать `0x` при выводе в шестнадцатеричной системе счисления; `e` – при выводе в научной нотации
*dec	Печатать в десятичной системе
hex	Печатать в шестнадцатеричной системе
oct	Печатать в восьмеричной системе

left	Добавлять символ заполнения справа от значения
right	Добавлять символ заполнения слева от значения
internal	Добавлять символ заполнения между знаком и значением
*fixed	Отображать число с плавающей точкой в десятичной нотации
scientific	Отображать число с плавающей точкой в научной нотации
flush	Сбросить буфер ostream
ends	Вставить нулевой символ, затем сбросить буфер ostream
endl	Вставить символ новой строки, затем сбросить буфер ostream
ws	Пропускать пробельные символы
// для этих манипуляторов требуется #include <ionamip>	
setfill(ch)	Заполнять пустое место символом ch
Setprecision(n)	Установить точность вывода числа с плавающей точкой равной n
setw(w)	Установить ширину поля ввода или вывода равной w
setbase(b)	Выводить целые числа по основанию b
* обозначает состояние потока по умолчанию	

20.10. Сильно типизированная библиотека

Библиотека iostream сильно типизирована. Например, попытка прочитать из объекта класса ostream или записать в объект класса istream помечается компилятором как

```
#include <iostream>
#include <fstream>
class Screen;

extern istream& operator>>( istream&, const Screen& );
extern void print( ostream& );
```

нарушение типизации. Так, если имеется набор объявлений:

```
ifstream inFile;
```

то следующие две инструкции приводят к нарушению типизации, обнаруживаемому во время компиляции:

```
int main()
{
   Screen myScreen;

   // ошибка: ожидается ostream&
   print( cin >> myScreen );

   // ошибка: ожидается оператор >>
   inFile << "ошибка: оператор вывода";
```

Средства ввода/вывода включены в состав стандартной библиотеки C++. В главе 20 библиотека `iostream` описана не полностью, в частности вопрос о создании определенных пользователем манипуляторов и буферных классов остался за рамками введения в язык. Мы сосредоточили внимание лишь на той части библиотеки `iostream`, которая имеет основополагающее значение для программного ввода/вывода.

Приложение

21. Обобщенные алгоритмы в алфавитном порядке

В этом Приложении мы рассмотрим все алгоритмы. Мы решили расположить их в алфавитном порядке (за небольшими исключениями), чтобы проще было найти нужный. Каждый алгоритм представлен в следующем виде: сначала описывается прототип функции, затем сам алгоритм, причем особое внимание уделяется интуитивно неочевидным особенностям, и, наконец, приводится пример программы, показывающий, как можно данный алгоритм использовать.

Первыми двумя аргументами всех обобщенных алгоритмов (естественно, не без исключений) является пара итераторов, обычно `first` и `last`, обозначающих диапазон элементов внутри контейнера или встроенного массива, над которым работает алгоритм. Этот диапазон (часто называемый *интервалом с включенной левой*

```
// следует читать: включая first и все последующие
// элементы до last, но не включая сам last
```

границей), как правило, записывается в виде:

```
[ first, last )
```

Это означает, что диапазон начинается с `first` и заканчивается `last`, однако сам элемент `last` *не* включается. Если

```
first == last
```

то говорят, что диапазон пуст.

К паре итераторов предъявляется такое требование: `last` должен быть достижим, если начать с `first` и последовательно применять оператор инкремента. Однако компилятор не может проверить выполнение данного ограничения. Если требование не будет выполнено, поведение программы не определено; обычно это заканчивается ее крахом и дампом памяти.

В объявлении каждого алгоритма подразумевается минимальная поддержка, которую должны обеспечить итераторы (краткое обсуждение пяти категорий итераторов см. в разделе 12.4). Например, алгоритм `find()`, реализующий однопроходный обход контейнера и выполняющий только чтение, требует итератора чтения `InputIterator`. Ему также можно передать одно- или двунаправленный итератор или итератор с произвольным доступом. Однако передача итератора записи приведет к ошибке. Не гарантируется, что подобные ошибки (при передаче итератора неподходящей категории) будут обнаружены компилятором, поскольку категории итераторов – это не сами типы, а лишь параметры, которыми конкретизируется шаблон функции.

Некоторые алгоритмы существуют в нескольких вариантах: в одном используется тот или иной встроенный оператор, а в другом – объект-функция или указатель на функцию, реализующие альтернативу этому оператору. Например, алгоритм `unique()` по умолчанию сравнивает соседние элементы контейнера с помощью оператора

равенства, определенного в классе, к которому данные элементы принадлежат. Но если в этом классе нет оператора равенства или мы хотим сравнивать элементы иным способом, то можем передать объект-функцию или указатель на функцию, поддерживающую нужную семантику. Есть и такие алгоритмы, которые выполняют похожие действия, но имеют различные имена. Так, имена предикатных версий алгоритмов всегда заканчиваются на `_if`, скажем `find_if()`. Например, есть вариант алгоритма `replace()`, где используется встроенный оператор равенства, и вариант с именем `replace_if()`, которому передается предикатный объект-функция или указатель на функцию-предикат.

Алгоритмы, модифицирующие контейнер, обычно также существуют в двух вариантах: один осуществляет модификацию по месту, а второй возвращает копию с внесенными изменениями. Так, есть алгоритмы `replace()` и `replace_copy()`. Однако вариант с копированием (его имя всегда содержит слово `_copy`) имеется не для каждого алгоритма, модифицирующего контейнер. К примеру, для алгоритмов `sort()` его нет. В таких случаях, если нужно, чтобы алгоритм работал с копией, мы должны создать ее самостоятельно и передать в качестве аргумента.

Для использования любого обобщенного алгоритма в программу необходимо включить заголовочный файл

```
#include <algorithm>
```

Для употребления любого из четырех численных алгоритмов: `adjacent_difference()`, `accumulate()`, `inner_product()` и `partial_sum()` – нужно включить также файл

```
#include <numeric>
```

Приведенные в этом Приложении примеры программ, в которых используются алгоритмы и различные контейнерные типы, отражают существующую на момент написания книги реализацию. Применение библиотеки ввода/вывода `iostream` следует соглашениям, установленным до принятия стандарта; скажем, в программу включается заголовочный файл `iostream.h`, а не `iostream`. Шаблоны не поддерживают аргументы по умолчанию. Чтобы программа работала на системе, имеющейся у вас, возможно, придется изменить некоторые объявления.

Другое, более подробное, чем в этой книге, описание обобщенных алгоритмов можно найти в работе [MUSSER96], правда, оно несколько отстает от окончательного варианта стандартной библиотеки C++.

Алгоритм accumulate()

```
template < class InputIterator, class Type >
Type accumulate(
   InputIterator first, InputIterator last,
   Type init );

template < class InputIterator, class Type,
           class BinaryOperation >
Type accumulate(
   InputIterator first, InputIterator last,
   Type init, BinaryOperation op );
```

Первый вариант accumulate() вычисляет сумму значений элементов последовательности из диапазона, ограниченного парой итераторов [first,last), с начальным значением, которое задано параметром init. Например, если дана последовательность {1,1,2,3,5,8} и начальное значение 0, то результатом работы алгоритма будет 20. Во втором варианте вместо оператора сложения к элементам применяется переданная бинарная операция. Если бы мы передали алгоритму accumulate() объект-функцию times<int> и начальное значение 1, то получили бы результат 240. accumulate() – это один из численных алгоритмов; для его использования в программу необходимо включить заголовочный файл <numeric>.

```
#include <numeric>
#include <list>
#include <functional>
#include <iostream.h>

/*
 * выход:
 * accumulate()
 *         работает с последовательностью {1,2,3,4}
 *         результат для сложения по умолчанию: 10
 *         результат для объекта-функции plus<int>: 10
 */

int main()
{
          int ia[] = { 1, 2, 3, 4 };
          list<int,allocator> ilist( ia, ia+4 );

          int ia_result = accumulate(&ia[0], &ia[4], 0);
          int ilist_res = accumulate(
         ilist.begin(), ilist.end(), 0, plus<int>() );

          cout << "accumulate()\n\t"
               << "работает с последовательностью {1,2,3,4}\n\t"
               << "результат для сложения по умолчанию: "
               << ia_result << "\n\t"
               << "результат для объекта-функции plus<int>: "
               << ilist_res
               << endl;

          return 0;
}
```

```
template < class InputIterator, class OutputIterator >
OutputIterator adjacent_difference(
   InputIterator first, InputIterator last,
   OutputIterator result );

template < class InputIterator, class OutputIterator >
           class BinaryOperation >
OutputIterator adjacent_difference(
   InputIterator first, InputIterator last,
```

Алгоритм adjacent_difference()

```
   OutputIterator result, BinaryOperation op );
```

Первый вариант `adjacent_difference()` создает новую последовательность, в которой значение каждого элемента, кроме первого, равно разности между текущим и предыдущим элементами исходной последовательности. Например, если дано {0,1,1,2,3,5,8}, то первым элементом новой последовательности будет копия: `0`. Вторым – разность первых двух элементов исходной последовательности: `1`. Третий элемент равен разности третьего и второго элементов: 1`-1`=0, и т.д. В результате мы получим последовательность {0,1,0,1,1,2,3}.

Во втором варианте разность соседних элементов вычисляется с помощью указанной бинарной операции. Возьмем ту же исходную последовательность и передадим объект-функцию `times<int>`. Как и раньше, первый элемент просто копируется. Второй элемент – это произведение первого и второго элементов исходной последовательности; он тоже равен `0`. Третий элемент – произведение второго и третьего элементов исходной последовательности: 1 * 1 = 1, и т.д. Результат – {0,1,2,6,15,40}.

В обоих вариантах итератор `OutputIterator` указывает на элемент, расположенный за последним элементом новой последовательности. `adjacent_difference()` – это один из численных алгоритмов, для его использования в программу необходимо включить заголовочный файл `<numeric>`.

```
#include <numeric>
#include <list>
#include <functional>
#include <iterator>
#include <iostream.h>

int main()
{
          int ia[] = { 1, 1, 2, 3, 5, 8 };

          list<int,allocator> ilist(ia, ia+6);
          list<int,allocator> ilist_result(ilist.size());

          adjacent_difference(ilist.begin(), ilist.end(),
                         ilist_result.begin() );

          // на выходе печатается:
     // 1 0 1 1 2 3
          copy( ilist_result.begin(), ilist_result.end(),
                ostream_iterator<int>(cout," "));
          cout << endl;

          adjacent_difference(ilist.begin(), ilist.end(),
                         ilist_result.begin(), times<int>() );

          // на выходе печатается:
          // 1 1 2 6 15 40
          copy( ilist_result.begin(), ilist_result.end(),
                ostream_iterator<int>(cout," "));
     cout << endl;

}
```

```
template < class ForwardIterator >
ForwardIterator
adjacent_find( ForwardIterator first, ForwardIterator last );

template < class ForwardIterator, class BinaryPredicate >
ForwardIterator
adjacent_find( ForwardIterator first,
               ForwardIterator last, Predicate pred );
```

Алгоритм adjacent_find()

`adjacent_find()` ищет первую пару одинаковых соседних элементов в диапазоне, ограниченном итераторами `[first,last)`. Если соседние дубликаты найдены, то алгоритм возвращает однонаправленный итератор, указывающий на первый элемент пары, в противном случае возвращается `last`. Например, если дана последовательность {0,1,1,2,2,4}, то будет найдена пара [1,1] и возвращен итератор, указывающий на первую единицу.

```
#include <algorithm>
#include <vector>
#include <iostream.h>
#include <assert.h>

class TwiceOver {
public:
          bool operator() ( int val1, int val2 )
               { return val1 == val2/2 ? true : false; }
};

int main()
{
          int ia[] = { 1, 4, 4, 8 };
          vector< int, allocator > vec( ia, ia+4 );

     int *piter;
          vector< int, allocator >::iterator iter;

          // piter указывает на ia[1]
          piter = adjacent_find( ia, ia+4 );
          assert( *piter == ia[ 1 ] );

          // iter указывает на vec[2]
          iter = adjacent_find( vec.begin(), vec.end(), TwiceOver() );
          assert( *iter == vec[ 2 ] );

          // пришли сюда: все хорошо
          cout << "ok: adjacent-find() завершился успешно!\n";

          return 0;
}
```

```
template < class ForwardIterator, class Type >
bool
binary_search( ForwardIterator first,
               ForwardIterator last, const Type &value );

template < class ForwardIterator, class Type >
bool
binary_search( ForwardIterator first,
               ForwardIterator last, const Type &value,
               Compare comp );
```

Алгоритм binary_search()

binary_search() ищет значение value в отсортированной последовательности, ограниченной парой итераторов [first,last). Если это значение найдено, возвращается true, иначе – false. В первом варианте предполагается, что контейнер отсортирован с помощью оператора “меньше”. Во втором варианте порядок определяется указанным объектом-функцией.

```
#include <algorithm>
#include <vector>
#include <assert.h>

int main()
{
          int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};
          vector< int, allocator > vec( ia, ia+12 );

          sort( &ia[0], &ia[12] );
          bool found_it = binary_search( &ia[0], &ia[12], 18 );
          assert( found_it == false );

     vector< int > vec( ia, ia+12 );
     sort( vec.begin(), vec.end(), greater<int>() );
          found_it = binary_search( vec.begin(), vec.end(),
                                    26, greater<int>() );
          assert( found_it == true );

}
```

```
template < class InputIterator, class OutputIterator >
OutputIterator
copy( InputIterator first1, InputIterator last,
      OutputIterator first2 )
```

Алгоритм copy()

copy() копирует последовательность элементов, ограниченную парой итераторов [first,last), в другой контейнер, начиная с позиции, на которую указывает first2. Алгоритм возвращает итератор, указывающий на элемент второго контейнера, следующий за последним вставленным. Например, если дана последовательность

```
int ia[] = {0, 1, 2, 3, 4, 5 };

// сдвинуть элементы влево на один, получится {1,2,3,4,5,5}
```

{0,1,2,3,4,5}, мы можем сдвинуть элементы на один влево с помощью такого вызова:

```
copy( ia+1, ia+6, ia );
```

copy() начинает копирование со второго элемента ia, копируя 1 в первую позицию, и так далее, пока каждый элемент не окажется в позиции на одну левее исходной.

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <iostream.h>

/* печатается:
   0 1 1 3 5 8 13
   сдвиг массива влево на 1:
   1 1 3 5 8 13 13
   сдвиг вектора влево на 2:
   1 3 5 8 13 8 13
*/

int main()
{
          int ia[] = { 0, 1, 1, 3, 5, 8, 13 };
          vector< int, allocator > vec( ia, ia+7 );
     ostream_iterator< int >  ofile( cout, " " );

     cout << "исходная последовательность элементов:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

          // сдвиг влево на один элемент
          copy( ia+1, ia+7, ia );

     cout << "сдвиг массива влево на 1:\n";
     copy( ia, ia+7, ofile ); cout << '\n';

          // сдвиг влево на два элемента
          copy( vec.begin()+2, vec.end(), vec.begin() );

     cout << "сдвиг вектора влево на 2:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

}
```

```
template < class BidirectionalIterator1,
           class BidirectionalIterator2 >
BidirectionalIterator2
copy_backward( BidirectionalIterator1 first,
               BidirectionalIterator1 last1,
               BidirectionalIterator2 last2 )
```

Алгоритм copy_backward()

copy_backward() ведет себя так же, как copy(), только элементы копируются в обратном порядке: копирование начинается с last1-1 и продолжается до first. Кроме того, элементы помещаются в целевой контейнер с конца, от позиции last2-1, пока не будет скопировано last1-first элементов.

Например, если дана последовательность {0,1,2,3,4,5}, мы можем скопировать последние три элемента (3,4,5) на место первых трех (0,1,2), установив first равным адресу значения 0, last1 – адресу значения 3, а last2 – адресу значения 5. Тогда

элемент 5 попадает на место элемента 2, элемент 4 – на место 1, а элемент 3 – на место

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <iostream.h>

class print_elements {
public:
   void operator()( string elem ) {
        cout << elem
             << ( _line_cnt++%8 ? " " : "\n\t" );
   }
   static void reset_line_cnt() { _line_cnt = 1; }

private:
   static int _line_cnt;
};

int print_elements::_line_cnt = 1;

/* печатается:
   исходный список строк:
   The light untonsured hair grained and hued like
   pale oak

   после copy_backward( begin+1, end-3, end ):
   The light untonsured hair light untonsured hair grained
   and hued
*/

int main()
{
   string sa[] = {
      "The", "light", "untonsured", "hair",
           "grained", "and", "hued", "like", "pale", "oak" };

   vector< string, allocator > svec( sa, sa+10 );

   cout << "исходный список строк:\n\t";
   for_each( svec.begin(), svec.end(), print_elements() );
   cout << "\n\n";

   copy_backward( svec.begin()+1, svec.end()-3, svec.end() );

   print_elements::reset_line_cnt();

   cout << "после copy_backward( begin+1, end-3, end ):\n";
   for_each( svec.begin(), svec.end(), print_elements() );
   cout << "\n";
```

0. В результате получим последовательность {3,4,5,3,4,5}.

```
}
```

Алгоритм count()

```
template < class InputIterator, class Type >
iterator_traits<InputIterator>::distance_type
count( InputIterator first,
       InputIterator last, const Type& value );
```

count() сравнивает каждый элемент со значением value в диапазоне, ограниченном парой итераторов [first,last), с помощью оператора равенства. Алгоритм возвращает число элементов, равных value. (Отметим, что в имеющейся у нас реализации стандартной библиотеки поддерживается более ранняя спецификация count().)

```
#include <algorithm>
#include <string>
#include <list>
#include <iterator>

#include <assert.h>
#include <iostream.h>
#include <fstream.h>

/*************************************************************************
* прочитанный текст:
Alice Emma has long flowing red hair. Her Daddy says
when the wind blows through her hair, it looks almost alive,
like a fiery bird in flight. A beautiful fiery bird, he tells her,
magical but untamed. "Daddy, shush, there is no such thing,"
she tells him, at the same time wanting him to tell her more.
Shyly, she asks, "I mean, Daddy, is there?"
*************************************************************************
          * программа выводит:
          *     count(): fiery встречается 2 раз(а)
*************************************************************************
*/

int main()
{

   ifstream infile( "alice_emma" );
   assert ( infile != 0 );

   list<string,allocator> textlines;

   typedef list<string,allocator>::difference_type diff_type;
   istream_iterator< string, diff_type > instream( infile ),
                     eos;

   copy( instream, eos, back_inserter( textlines ));

   string search_item( "fiery" );


         /*************************************************************
         ********
    * примечание: ниже показан интерфейс count(), принятый в
    *             стандартной библиотеке. В текущей реализации
         библиотеки
    *       от RogueWave поддерживается более ранняя версия, в которой
    *       типа distance_type еще не было, так что count()
    *       возвращала результат в последнем аргументе
    *
    * вот как должен выглядеть вызов:
    *
    * typedef iterator_traits<InputIterator>::
    *     distance_type dis_type;
    *
    * dis_type elem_count;
    * elem_count = count( textlines.begin(), textlines.end(),
    *                     search_item );

         *************************************************************
         *********

   int elem_count = 0;
   list<string,allocator>::iterator
        ibegin = textlines.begin(),
        iend   = textlines.end();

   // устаревшая форма count()
   count( ibegin, iend, search_item, elem_count );

   cout << "count(): " << search_item
        << " встречается "  << elem_count << " раз(а)\n";
```

```
}
```

```
template < class InputIterator, class Predicate >
iterator_traits<InputIterator>::distance_type
count_if( InputIterator first,
```

Алгоритм count_if()

```
          InputIterator last, Predicate pred );
```

`count_if()` применяет предикат `pred` к каждому элементу из диапазона, ограниченного парой итераторов `[first,last)`. Алгоритм сообщает, сколько раз предикат оказался равным `true`.

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

class Even {
public:
          bool operator()( int val )
          { return val%2 ? false : true; }
};

int main()
{
   int ia[] = {0,1,1,2,3,5,8,13,21,34};
   list< int,allocator > ilist( ia, ia+10 );

   /*
    * не поддерживается в текущей реализации
    ****************************************************
    typedef
       iterator_traits<InputIterator>::distance_type
            distance_type;

            distance_type ia_count, list_count;

            // счетчик четных элементов: 4
       ia_count = count_if( &ia[0], &ia[10], Even() );
            list_count = count_if( ilist.begin(), ilist_end(),
                                   bind2nd(less<int>(),10) );
    ****************************************************
    */
   int ia_count = 0;
   count_if( &ia[0], &ia[10], Even(), ia_count );

   // печатается:
   //   count_if(): есть 4 четных элемент(а).

   cout << "count_if(): есть "
        << ia_count << " четных элемент(а).\n";

   int list_count = 0;
   count_if( ilist.begin(), ilist.end(),
             bind2nd(less<int>(),10), list_count );

   // печатается:
   //   count_if(): есть 7 элемент(ов), меньших 10.

   cout << "count_if(): есть "
        << list_count
        << " элемент(ов), меньших 10.\n";

}
```

```
template< class InputIterator1, class InputIterator2 >
bool
equal( InputIterator1 first1,
       InputIterator1 last, InputIterator2 first2 );

template< class InputIterator1, class InputIterator2,
          class BinaryPredicate >
bool
equal( InputIterator1 first1, InputIterator1 last,
       InputIterator2 first2, BinaryPredicate pred );
```

Алгоритм equal()

`equal()` возвращает `true`, если обе последовательности одинаковы в диапазоне, ограниченном парой итераторов `[first,last)`. Если вторая последовательность содержит дополнительные элементы, они игнорируются. Чтобы убедиться в

```
if ( vec1.size() == vec2.size() &&
```

тождественности данных последовательностей, необходимо написать:

```
     equal( vec1.begin(), vec1.end(), vec2.begin() );
```

или воспользоваться оператором равенства, определенном в классе самого контейнера: `vec1 == vec2`. Если второй контейнер содержит меньше элементов, чем первый, и алгоритму приходится просматривать элементы за концом контейнера, то поведение программы не определено. По умолчанию для сравнения применяется оператор равенства в классе элементов контейнера, а во втором варианте алгоритма – указанный предикат `pred`.

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

class equal_and_odd{
public:
          bool
          operator()( int val1, int val2 )
     {
        return ( val1 == val2 &&
               ( val1 == 0 || val1 % 2 ))
                        ? true : false;
          }
};

int main()
{
   int ia[] =  { 0,1,1,2,3,5,8,13 };
   int ia2[] = { 0,1,1,2,3,5,8,13,21,34 };

   bool res;

   // true: обе последовательности совпадают до длины ia
   // печатается: int ia[7] равно int ia2[9]? Да.

   res = equal( &ia[0], &ia[7], &ia2[0] );
   cout << "int ia[7] равно int ia2[9]? "
        << ( res ? "Да" : "Нет" ) << ".\n";

   list< int, allocator > ilist(  ia,  ia+7  );
   list< int, allocator > ilist2( ia2, ia2+9 );

   // печатается: список ilist равен ilist2? Да.

   res = equal( ilist.begin(), ilist.end(), ilist2.begin() );
   cout << "список ilist равен ilist2? "
        << ( res ? "Да" : "Нет" ) << ".\n";

   // false: 0, 2, 8 не являются равными и нечетными
   // печатается: список ilist equal_and_odd() ilist2? Нет.

   res = equal( ilist.begin(), ilist.end(),
                ilist2.begin(), equal_and_odd() );

   cout << "список ilist equal_and_odd() ilist2? "
        << ( res ? "Да" : "Нет" ) << ".\n";

   return 0;
}
```

```
template< class ForwardIterator, class Type >
pair< ForwardIterator, ForwardIterator >
equal_range( ForwardIterator first,
             ForwardIterator last, const Type &value );

template< class ForwardIterator, class Type, class Compare >
pair< ForwardIterator, ForwardIterator >
equal_range( ForwardIterator first,
             ForwardIterator last, const Type &value,
```

Алгоритм equal_range()

```
             Compare comp );
```

`equal_range()` возвращает пару итераторов: первый представляет значение итератора, возвращаемое алгоритмом `lower_bound()`, второй – алгоритмом `upper_bound()`. (О семантике этих алгоритмов рассказано в их описаниях.) Например, дана последовательность:

```
int ia[] = {12,15,17,19,20,22,23,26,29,35,40,51};
```

Обращение к `equal_range()` со значением `21` возвращает пару итераторов, в которой оба указывают на значение `22`. Обращение же со значением `22` возвращает пару итераторов, где `first` указывает на `22`, а `second` – на `23`. В первом варианте при сравнении используется оператор “меньше”, определенный для типа элементов контейнера; во втором – предикат `comp`.

```
#include <algorithm>
#include <vector>
#include <utility>
#include <iostream.h>

/* печатается:
   последовательность элементов массива после сортировки:
   12 15 17 19 20 22 23 26 29 35 40 51

   результат equal_range при поиске значения 23:
        *ia_iter.first: 23      *ia_iter.second: 26

   результат equal_range при поиске отсутствующего значения 21:
        *ia_iter.first: 22      *ia_iter.second: 22

   последовательность элементов вектора после сортировки:
   51 40 35 29 26 23 22 20 19 17 15 12

   результат equal_range при поиске значения 26:
        *ivec_iter.first: 26    *ivec_iter.second: 23

   результат equal_range при поиске отсутствующего значения 21:
        *ivec_iter.first: 20    *ivec_iter.second: 20
*/
int main()
{
   int ia[] = { 29,23,20,22,17,15,26,51,19,12,35,40 };
   vector< int, allocator > ivec( ia, ia+12 );
   ostream_iterator< int >  ofile( cout, " " );

   sort( &ia[0], &ia[12] );

   cout << "последовательность элементов массива после сортировки:\n";
   copy( ia, ia+12, ofile ); cout << "\n\n";

   pair< int*,int* > ia_iter;
   ia_iter = equal_range( &ia[0], &ia[12], 23 );

   cout << "результат equal_range при поиске значения 23:\n\t"
        << "*ia_iter.first: "  << *ia_iter.first << "\t"
        << "*ia_iter.second: " << *ia_iter.second << "\n\n";

   ia_iter = equal_range( &ia[0], &ia[12], 21 );

   cout << "результат equal_range при поиске "
        << "отсутствующего значения 21:\n\t"
        << "*ia_iter.first: "  << *ia_iter.first << "\t"
        << "*ia_iter.second: " << *ia_iter.second << "\n\n";

   sort( ivec.begin(), ivec.end(), greater<int>() );

   cout << "последовательность элементов вектора после сортировки:\n";
   copy( ivec.begin(), ivec.end(), ofile ); cout << "\n\n";

   typedef vector< int, allocator >::iterator iter_ivec;
   pair< iter_ivec, iter_ivec > ivec_iter;

   ivec_iter = equal_range( ivec.begin(), ivec.end(), 26,
               greater<int>() );

   cout << "результат equal_range при поиске значения 26:\n\t"
        << "*ivec_iter.first: "  << *ivec_iter.first << "\t"
             << "*ivec_iter.second: " << *ivec_iter.second
        << "\n\n";

   ivec_iter = equal_range( ivec.begin(), ivec.end(), 21,
               greater<int>() );

   cout << "результат equal_range при поиске отсутствующего значения
          21:\n\t"
        << "*ivec_iter.first: "  << *ivec_iter.first << "\t"
             << "*ivec_iter.second: " << *ivec_iter.second
        << "\n\n";
```

```
}
```

```
template< class ForwardIterator, class Type >
void
fill( ForwardIterator first,
```

Алгоритм fill()

```
      ForwardIterator last, const Type& value );
```

`fill()` помещает копию значения `value` в каждый элемент диапазона, ограниченного парой итераторов `[first,last)`.

```
#include <algorithm>
#include <list>
#include <string>
#include <iostream.h>

/* печатается:
   исходная последовательность элементов массива:
   0 1 1 2 3 5 8

   массив после fill(ia+1,ia+6):
   0 9 9 9 9 9 8

   исходная последовательность элементов списка:
   c eiffel java ada perl

   список после fill(++ibegin,--iend):
   c c++ c++ c++ perl
*/

int main()
{
   const int value = 9;
   int ia[]  = { 0, 1, 1, 2, 3, 5, 8 };
   ostream_iterator< int > ofile( cout, " " );

   cout << "исходная последовательность элементов массива:\n";
   copy( ia, ia+7, ofile ); cout << "\n\n";

   fill( ia+1, ia+6, value );

   cout << "массив после fill(ia+1,ia+6):\n";
   copy( ia, ia+7, ofile ); cout << "\n\n";

   string the_lang( "c++" );
   string langs[5] = { "c", "eiffel", "java", "ada", "perl" };

   list< string, allocator > il( langs, langs+5 );
   ostream_iterator< string > sofile( cout, " " );

   cout << "исходная последовательность элементов списка:\n";
   copy( il.begin(), il.end(), sofile ); cout << "\n\n";

   typedef list<string,allocator>::iterator iterator;

   iterator ibegin = il.begin(), iend = il.end();
   fill( ++ibegin, --iend, the_lang );

   cout << "список после fill(++ibegin,--iend):\n";
   copy( il.begin(), il.end(), sofile ); cout << "\n\n";

}
```

Алгоритм fill_n()

```
template< class ForwardIterator, class Size, class Type >
void
fill_n( ForwardIterator first,
        Size n, const Type& value );
```

`fill_n()` присваивает `count` элементам из диапазона `[first,first+count)` значение `value`.

```
#include <algorithm>
#include <vector>
#include <string>
#include <iostream.h>

class print_elements {
public:
   void operator()( string elem ) {
      cout << elem
           << ( _line_cnt++%8 ? " " : "\n\t" );
   }
   static void reset_line_cnt() { _line_cnt = 1; }

private:
   static int _line_cnt;
};

int print_elements::_line_cnt = 1;

/* печатается:
исходная последовательность элементов массива:
0 1 1 2 3 5 8

массив после fill_n( ia+2, 3, 9 ):
0 1 9 9 9 5 8
исходная последовательность строк:
        Stephen closed his eyes to hear his boots
        crush crackling wrack and shells

последовательность после применения fill_n():
        Stephen closed his xxxxx xxxxx xxxxx xxxxx xxxxx
        xxxxx crackling wrack and shells
*/

int main()
{
   int value = 9; int count = 3;
   int ia[]  = { 0, 1, 1, 2, 3, 5, 8 };
   ostream_iterator< int > iofile( cout, " " );

   cout << "исходная последовательность элементов массива:\n";
   copy( ia, ia+7, iofile ); cout << "\n\n";

   fill_n( ia+2, count, value );

   cout << "массив после fill_n( ia+2, 3, 9 ):\n";
   copy( ia, ia+7, iofile ); cout << "\n\n";

   string replacement( "xxxxx" );
   string sa[] = { "Stephen", "closed", "his", "eyes", "to",
         "hear", "his", "boots", "crush", "crackling",
         "wrack", "and", "shells" };

   vector< string, allocator > svec( sa, sa+13 );

   cout << "исходная последовательность строк:\n\t";
   for_each( svec.begin(), svec.end(), print_elements() );
   cout << "\n\n";

   fill_n( svec.begin()+3, count*2, replacement );

   print_elements::reset_line_cnt();

   cout << "последовательность после применения fill_n():\n\t";
   for_each( svec.begin(), svec.end(), print_elements() );
   cout << "\n";
```

```
}
```

```
template< class InputIterator, class T >
InputIterator
find( InputIterator first,
```

Алгоритм find()

```
      InputIterator last, const T &value );
```

Элементы из диапазона, ограниченного парой итераторов [first,last), сравниваются со значением value с помощью оператора равенства, определенного для типа элементов контейнера. Как только соответствие найдено, поиск прекращается. find() возвращает итератор типа InputIterator, указывающий на найденный

```
#include <algorithm>
#include <iostream.h>
#include <list>
#include <string>

int main()
{
          int array[ 17 ] = { 7,3,3,7,6,5,8,7,2,1,3,8,7,3,8,4,3 };

          int elem = array[ 9 ];
          int *found_it;

          found_it = find( &array[0], &array[17], elem );

          // печатается: поиск первого вхождения 1 найдено!

          cout << "поиск первого вхождения "
               << elem << "\t"
               << ( found_it ? "найдено!\n" : "не найдено!\n" );

     string beethoven[] = {
             "Sonata31", "Sonata32", "Quartet14", "Quartet15",
             "Archduke", "Symphony7" };

          string s_elem( beethoven[ 1 ] );

     list< string, allocator > slist( beethoven, beethoven+6 );
          list< string, allocator >::iterator iter;

          iter = find( slist.begin(), slist.end(), s_elem );

          // печатается: поиск первого вхождения Sonata32 найдено!

          cout << "поиск первого вхождения "
               << s_elem << "\t"
               << ( found_it ? "найдено!\n" : "не найдено!\n" );
```

элемент; в противном случае возвращается last.

```
}
```

```
template< class InputIterator, class Predicate >
InputIterator
find_if( InputIterator first,
```

Алгоритм find_if()

```
         InputIterator last, Predicate pred );
```

К каждому элементу из диапазона `[first,last)` последовательно применяется предикат `pred`. Если он возвращает `true`, поиск прекращается. `find_if()` возвращает итератор типа `InputIterator`, указывающий на найденный элемент; в противном случае возвращается `last`.

```
#include <algorithm>
#include <list>
#include <set>
#include <string>
#include <iostream.h>

// альтернатива оператору равенства
// возвращает true, если строка содержится в объекте-члене FriendSet
class OurFriends {     // наши друзья
public:
    bool operator()( const string& str ) {
              return ( friendset.count( str ));
    }

    static void
    FriendSet( const string *fs, int count ) {
              copy( fs, fs+count,
             inserter( friendset, friendset.end() ));
    }

private:
    static set< string, less<string>, allocator > friendset;
};

set< string, less<string>, allocator > OurFriends::friendset;

int main()
{
    string Pooh_friends[] = { "Пятачок", "Тигра", "Иа-Иа"  };
    string more_friends[] = { "Квазимодо", "Чип", "Пятачок" };
    list<string,allocator> lf( more_friends, more_friends+3 );

    // заполнить список друзей Пуха
    OurFriends::FriendSet( Pooh_friends, 3 );

    list<string,allocator>::iterator our_mutual_friend;
    our_mutual_friend =
               find_if( lf.begin(), lf.end(), OurFriends());

    // печатается:
    //   Представьте-ка, наш друг Пятачок - также друг Пуха.
    if ( our_mutual_friend != lf.end() )
         cout << "Представьте-ка, наш друг "
              << *our_mutual_friend
              << " также друг Пуха.\n";

    return 0;
}
```

```
template< class ForwardIterator1, class ForwardIterator2 >
ForwardIterator1
find_end( ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2 );

template< class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate >
ForwardIterator1
find_end( ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2,
```

Алгоритм find_end()

```
          BinaryPredicate pred );
```

В последовательности, ограниченной итераторами [first1,last1), ведется поиск последнего вхождения последовательности, ограниченной парой [first2,last2). Например, если первая последовательность – это Mississippi, а вторая – ss, то find_end() возвращает итератор, указывающий на первую s во втором вхождении ss. Если вторая последовательность не входит в первую, то возвращается last1. В первом варианте используется оператор равенства, определенный для типа элементов контейнера, а во втором – бинарный предикат, переданный пользователем.

```
#include <algorithm>
#include <vector>
#include <iostream.h>
#include <assert.h>

int main()
{
          int array[ 17 ]   = { 7,3,3,7,6,5,8,7,2,1,3,7,6,3,8,4,3 };
          int subarray[ 3 ] = { 3, 7, 6 };

          int *found_it;

          // find найти последнее вхождение последовательности 3,7,6
          // в массив и вернуть адрес первого ее элемента ...

     found_it = find_end( &array[0],    &array[17],
                     &subarray[0], &subarray[3] );

          assert( found_it == &array[10] );

          vector< int, allocator > ivec( array, array+17 );
          vector< int, allocator > subvec( subarray, subarray+3 );
          vector< int, allocator >::iterator found_it2;

          found_it2 = find_end( ivec.begin(),   ivec.end(),
                          subvec.begin(), subvec.end(),
                          equal_to<int>() );

          assert( found_it2 == ivec.begin()+10 );

          cout << "ok: find_end правильно вернула начало "
               << "последнего вхождения последовательности: 3,7,6!\n";

}
```

```
template< class ForwardIterator1, class ForwardIterator2 >
ForwardIterator1
find_first_of( ForwardIterator1 first1, ForwardIterator1 last1,
               ForwardIterator2 first2, ForwardIterator2 last2 );

template< class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate >
ForwardIterator1
find_first_of( ForwardIterator1 first1, ForwardIterator1 last1,
               ForwardIterator2 first2, ForwardIterator2 last2,
               BinaryPredicate pred );
```

Алгоритм find_first_of()

Последовательность, ограниченная парой [first2,last2), содержит элементы, поиск которых ведется в последовательности, ограниченной итераторами [first1,last1). Допустим, нужно найти первую гласную в последовательности символов synesthesia. Для этого определим вторую последовательность как aeiou. find_first_of() возвращает итератор, указывающий на первое вхождение любого элемента

последовательности гласных букв, в данном случае e. Если же первая последовательность не содержит ни одного элемента из второй, то возвращается last1. В первом варианте используется оператор равенства, определенный для типа элементов

```
#include <algorithm>
#include <vector>
#include <string>
#include <iostream.h>

int main()
{
          string s_array[] = { "Ee", "eE", "ee", "Oo", "oo", "ee" };

          // возвращает первое вхождение "ee" -- &s_array[2]
          string to_find[] = { "oo", "gg", "ee" };

          string *found_it =
       find_first_of( s_array, s_array+6,
                      to_find, to_find+3 );
          // печатается:
          // найдено: ee
          //        &s_array[2]:    0x7fff2dac
          //        &found_it:      0x7fff2dac

          if ( found_it != &s_array[6] )
             cout << "найдено: "      << *found_it     << "\n\t"
               << "&s_array[2]:\t" << &s_array[2]   << "\n\t"
               << "&found_it:\t"   << found_it      << "\n\n";

          vector< string, allocator > svec( s_array, s_array+6);
          vector< string, allocator > svec_find( to_find, to_find+2 );

          // возвращает вхождение "oo" -- svec.end()-2
          vector< string, allocator >::iterator found_it2;

          found_it2 = find_first_of(
                     svec.begin(), svec.end(),
                     svec_find.begin(), svec_find.end(),
                     equal_to<string>() );

          // печатает:
          // тоже найдено: oo
          //         &svec.end()-2:  0x100067b0
          //         &found_it2:     0x100067b0

          if ( found_it2 != svec.end() )
             cout << "тоже найдено: "   << *found_it2   << "\n\t"
               << "&svec.end()-2:\t" << svec.end()-2 << "\n\t"
               << "&found_it2:\t"    << found_it2    << "\n";
```

контейнера, а во втором – бинарный предикат pred.

```
}
```

Алгоритм for_each()

```
template< class  InputIterator, class Function >
Function
for_each( InputIterator first,
          InputIterator last, Function func );
```

for_each() применяет объект-функцию func к каждому элементу в диапазоне [first,last). func не может изменять элементы, поскольку итератор записи не гарантирует поддержки присваивания. Если же модификация необходима, следует воспользоваться алгоритмом transform(). func может возвращать значение, но оно

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

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

int main()
{
          vector< int, allocator > ivec;

          for ( int ix = 0; ix < 10; ix++ )
                ivec.push_back( ix );

          void (*pfi)( int ) = print_elements;
          for_each( ivec.begin(), ivec.end(), pfi );

          return 0;
```

игнорируется.

```
}
```

```
template< class ForwardIterator, class Generator >
void
generate( ForwardIterator first,
```

Алгоритм generate()

```
          ForwardIterator last, Generator gen );
```

generate() заполняет диапазон, ограниченный парой итераторов [first,last), путем последовательного вызова gen, который может быть объектом-функцией или указателем на функцию.

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

int odd_by_twos() {
          static int seed = -1;
          return seed += 2;
}

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

int main()
{
          list< int, allocator > ilist( 10 );
          void (*pfi)( int ) = print_elements;

          generate( ilist.begin(), ilist.end(), odd_by_twos );

          // печатается:
          // элементы в списке, первый вызов:
          // 1 3 5 7 9 11 13 15 17 19

          cout << "элементы в списке, первый вызов:\n";
          for_each( ilist.begin(), ilist.end(), pfi );
          generate( ilist.begin(), ilist.end(), odd_by_twos );

          // печатается:
          // элементы в списке, второй вызов:
          // 21 23 25 27 29 31 33 35 37 39
          cout << "\n\nэлементы в списке, второй вызов:\n";
          for_each( ilist.begin(), ilist.end(), pfi );

          return 0;
}
```

```
template< class OutputIterator,
          class Size, class Generator >
void
```

Алгоритм generate_n()

```
generate_n( OutputIterator first, Size n, Generator gen );
```

generate_n() заполняет последовательность, начиная с first, n раз вызывая gen, который может быть объектом-функцией или указателем на функцию.

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

class even_by_twos {
public:
          even_by_twos( int seed = 0 ) : _seed( seed ){}
          int operator()() { return _seed += 2; }
private:
          int _seed;
};

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

int main()
{
    list< int, allocator > ilist( 10 );
    void (*pfi)( int ) = print_elements;

    generate_n( ilist.begin(), ilist.size(), even_by_twos() );

    // печатается:
    // generate_n с even_by_twos():
    // 2 4 6 8 10 12 14 16 18 20

   cout << "generate_n с even_by_twos():\n";
   for_each( ilist.begin(), ilist.end(), pfi ); cout << "\n";
   generate_n( ilist.begin(), ilist.size(), even_by_twos( 100 ) );

   // печатается:
   // generate_n с even_by_twos( 100 ):
   // 102 104 106 108 110 112 114 116 118 120

   cout << "generate_n с even_by_twos( 100 ):\n";
   for_each( ilist.begin(), ilist.end(), pfi );

}
```

```
template< class InputIterator1, class InputIterator2 >
bool
includes( InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2 );

template< class InputIterator1, class InputIterator2,
          class Compare >
bool
includes( InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          Compare comp );
```

Алгоритм includes()

includes() проверяет, каждый ли элемент последовательности [first1,last1) входит в последовательность [first2,last2). Первый вариант предполагает, что

последовательности отсортированы в порядке, определяемом оператором “меньше”;

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

int main()
{
          int ia1[] = { 13, 1, 21, 2, 0, 34, 5, 1, 8, 3, 21, 34 };
          int ia2[] = { 21, 2, 8, 3, 5, 1 };

          // алгоритму includes следует передавать отсортированные
          контейнеры
          sort( ia1, ia1+12 ); sort( ia2, ia2+6 );

          // печатает: каждый элемент ia2 входит в ia1? Да

          bool res = includes( ia1, ia1+12, ia2, ia2+6 );
          cout << "каждый элемент ia2 входит в ia1? "
               << (res ? "Да" : "Нет") << endl;

          vector< int, allocator > ivect1( ia1, ia1+12 );
          vector< int, allocator > ivect2( ia2, ia2+6 );

          // отсортирован в порядке убывания
          sort( ivect1.begin(), ivect1.end(), greater<int>() );
          sort( ivect2.begin(), ivect2.end(), greater<int>() );

          res = includes( ivect1.begin(), ivect1.end(),
                    ivect2.begin(), ivect2.end(),
                    greater<int>() );

          // печатает: каждый элемент ivect2 входит в ivect1? Да

          cout << "каждый элемент ivect2 входит в ivect1? "
               << (res ? "Да" : "Нет") << endl;
```

второй – что порядок задается параметром-типом comp.

```
}
```

Алгоритм inner_product()

```
template< class InputIterator1, class InputIterator2
          class Type >
Type
inner_product(
   InputIterator1 first1, InputIterator1 last,
   InputIterator2 first2, Type init );

template< class InputIterator1, class InputIterator2
          class Type,
          class BinaryOperation1, class BinaryOperation2 >
Type
inner_product(
   InputIterator1 first1, InputIterator1 last,
   InputIterator2 first2, Type init,
   BinaryOperation1 op1, BinaryOperation2 op2 );
```

Первый вариант суммирует произведения соответственных членов обеих последовательностей и прибавляет результат к начальному значению `init`. Первая последовательность ограничена итераторами `[first1,last1)`, вторая начинается с `first2` и обходится синхронно с первой. Например, если даны последовательности {2,3,5,8} и {1,2,3,4}, то результат вычисляется следующим образом:

```
2*1 + 3*2 + 5*3 + 8*4
```

Если начальное значение равно `0`, алгоритм вернет `55`.

Во втором варианте вместо сложения используется бинарная операция `op1`, а вместо умножения – бинарная операция `op1`. Например, если для приведенных выше последовательностей применить вычитание в качестве `op1` и сложение в качестве `op2`, то результат будет вычисляться так:

```
(2+1) - (3+2) - (5+3) - (8+4)
```

`inner_product()` – это один из численных алгоритмов. Для его использования в программу необходимо включить заголовочный файл `<numeric>`.

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

int main()
{
          int ia[] =  { 2, 3, 5, 8 };
          int ia2[] = { 1, 2, 3, 4 };

          // перемножить пары элементов из обоих массивов,
          // сложить и добавить начальное значение: 0

          int res = inner_product( &ia[0], &ia[4], &ia2[0], 0 );

          // печатает: скалярное произведение массивов: 55
          cout << "скалярное произведение массивов: "
               << res << endl;

          vector<int, allocator> vec(  ia,  ia+4 );
          vector<int, allocator> vec2( ia2, ia2+4 );

          // сложить пары элементов из обоих векторов,
          // вычесть из начального значения: 0

          res = inner_product( vec.begin(), vec.end(),
                     vec2.begin(), 0,
                     minus<int>(), plus<int>() );

          // печатает: скалярное произведение векторов: -28
          cout << "скалярное произведение векторов: "
               << res << endl;

          return 0;
}
```

```
template< class BidirectionalIterator >
void
inplace_merge( BidirectionalIterator first,
               BidirectionalIterator middle,
               BidirectionalIterator last );

template< class BidirectionalIterator, class Compare >
void
inplace_merge( BidirectionalIterator first,
               BidirectionalIterator middle,
               BidirectionalIterator last, Compare comp );
```

Алгоритм inplace_merge()

`inplace_merge()` объединяет две соседние отсортированные последовательности, ограниченные парами итераторов `[first,middle)` и `[middle,last)`. Результирующая последовательность затирает исходные, начиная с позиции `first`. В первом варианте

для упорядочения элементов используется оператор “меньше”, определенный для типа

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

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

/*
 * печатает:
ia разбит на два отсортированных подмассива:
12 15 17 20 23 26 29 35 40 51 10 16 21 41 44 54 62 65 71 74

ia inplace_merge:
10 12 15 16 17 20 21 23 26 29 35 40 41 44 51 54 62 65 71 74

ivec разбит на два отсортированных подвектора:
51 40 35 29 26 23 20 17 15 12 74 71 65 62 54 44 41 21 16 10

ivec inplace_merge:
74 71 65 62 54 51 44 41 40 35 29 26 23 21 20 17 16 15 12 10
*/

int main()
{
          int ia[] = { 29,23,20,17,15,26,51,12,35,40,
               74,16,54,21,44,62,10,41,65,71 };

          vector< int, allocator > ivec( ia, ia+20 );
        void (*pfi)( int ) = print_elements;

          // отсортировать обе подпоследовательности
          sort( &ia[0], &ia[10] );
          sort( &ia[10], &ia[20] );

          cout << "ia разбит на два отсортированных подмассива: \n";
     for_each( ia, ia+20, pfi ); cout << "\n\n";

          inplace_merge( ia, ia+10, ia+20 );

          cout << "ia inplace_merge:\n";
     for_each( ia, ia+20, pfi ); cout << "\n\n";

          sort( ivec.begin(),    ivec.begin()+10, greater<int>() );
          sort( ivec.begin()+10, ivec.end(),      greater<int>() );

          cout << "ivec разбит на два отсортированных подвектора: \n";
     for_each( ivec.begin(), ivec.end(), pfi ); cout << "\n\n";

          inplace_merge( ivec.begin(), ivec.begin()+10,
                    ivec.end(),   greater<int>() );

          cout << "ivec inplace_merge:\n";
     for_each( ivec.begin(), ivec.end(), pfi ); cout << endl;
```

элементов контейнера, во втором – операция сравнения, переданная программистом.

```
}
```

```
template< class ForwardIterator1, class ForwardIterator2 >
void
```

Алгоритм iter_swap()

```
iter_swap( ForwardIterator1 a, ForwardIterator2 b );
```

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

int main()
{
          int ia[]  = { 5, 4, 3, 2, 1, 0 };
          list< int,allocator > ilist( ia, ia+6 );

          typedef list< int, allocator >::iterator iterator;
          iterator iter1 = ilist.begin(),iter2,
           iter_end = ilist.end();

          // отсортировать список "пузырьком" ...
          for ( ; iter1 != iter_end; ++iter1 )
                for ( iter2 = iter1; iter2 != iter_end; ++iter2 )
              if ( *iter2 < *iter1 )
                   iter_swap( iter1, iter2 );

          // печатается:
          // ilist после сортировки "пузырьком" с помощью iter_swap():
    // { 0 1 2 3 4 5 }

          cout << "ilist после сортировки "пузырьком" с помощью
          iter_swap(): { ";
          for ( iter1 = ilist.begin(); iter1 != iter_end; ++iter1 )
                cout << *iter1 << " ";
       cout << "}\n";

          return 0;
```

`iter_swap()` обменивает значения элементов, на которые указывают итераторы `a` и `b`.

```
}
```

```
template< class InputIterator1, class InputIterator2 >
bool
lexicographical_compare(
   InputIterator1 first1, InputIterator1 last1,
   InputIterator1 first2, InputIterator2 last2 );

template< class InputIterator1, class InputIterator2,
          class Compare >
bool
lexicographical_compare(
   InputIterator1 first1, InputIterator1 last1,
   InputIterator1 first2, InputIterator2 last2,
```

Алгоритм lexicographical_compare()

```
   Compare comp );
```

`lexicographical_compare()` сравнивает соответственные пары элементов из двух последовательностей, ограниченных диапазонами `[first1, last1)` и `[first2, last2)`. Сравнение продолжается, пока не будет найдена первая пара различных элементов, не достигнута пара `[last1, last2]` или хотя бы один из элементов `last1` или `last2` (если последовательности имеют разные длины). При обнаружении первой пары различных элементов алгоритм возвращает:

- если меньше элемент первой последовательности, то `true`, иначе `false`;
- если `last1` достигнут, а `last2` нет, то `true`;
- если `last2` достигнут, а `last1` нет, то `false`;
- если достигнуты и `last1`, и `last2` (т.е. все элементы одинаковы), то `false`. Иными словами, первая последовательность лексикографически не меньше второй.

```
string arr1[] = { "Piglet", "Pooh", "Tigger" };
```

Например, даны такие последовательности:

```
string arr2[] = { "Piglet", "Pooch", "Eeyore" };
```

В них первая пара элементов одинакова, а вторая различна. `Pooh` считается больше, чем `Pooch`, так как `c` лексикографически меньше `h` (такой способ сравнения применяется при составлении словарей). В этом месте алгоритм заканчивается (третья пара элементов не сравнивается). Результатом сравнения будет `false`.

Во втором варианте алгоритма вместо оператора сравнения используется предикатный объект:

```
#include <algorithm>
#include <list>
#include <string>
#include <assert.h>
#include <iostream.h>

class size_compare {
public:
          bool operator()( const string &a, const string &b ) {
               return a.length() <= b.length();
          }
};

int main()
{
          string arr1[] = { "Piglet", "Pooh", "Tigger" };
          string arr2[] = { "Piglet", "Pooch", "Eeyore" };

          bool res;

          // на втором элементе получаем false
          // Pooch меньше Pooh
          // на третьем элементе тоже получили бы false

          res = lexicographical_compare( arr1, arr1+3,
                                    arr2, arr2+3 );

          assert( res == false );

          // получаем true: длина каждого элемента ilist2
          // меньше либо равна длине соответственного
          // элемента ilist1

          list< string, allocator > ilist1( arr1, arr1+3 );
          list< string, allocator > ilist2( arr2, arr2+3 );

          res = lexicographical_compare(
                   ilist1.begin(), ilist1.end(),
                   ilist2.begin(), ilist2.end(), size_compare() );

          assert( res == true );

          cout << "ok: lexicographical_compare завершился успешно!\n";
}
```

Алгоритм lower_bound()

```
template< class ForwardIterator, class Type >
ForwardIterator
lower_bound( ForwardIterator first,
             ForwardIterator last, const Type &value );


template< class ForwardIterator, class Type, class Compare >
ForwardIterator
lower_bound( ForwardIterator first,
             ForwardIterator last, const Type &value,
             class Compare );
```

`lower_bound()` возвращает итератор, указывающий на первую позицию в отсортированной последовательности, ограниченной диапазоном `[first,last)`, в которую можно вставить значение `value`, не нарушая упорядоченности. В этой позиции находится значение, большее либо равное `value`. Например, если дана такая последовательность:

```
int ia = = {12,15,17,19,20,22,23,26,29,35,40,51};
```

то обращение к `lower_bound()` с аргументом `value=21` возвращает итератор, указывающий на `23`. Обращение с аргументом `22` возвращает тот же итератор. В первом варианте алгоритма используется оператор “меньше”, определенный для типа элементов контейнера, а во втором для упорядочения элементов применяется объект `comp`.

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

int main()
{
          int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};
          sort( &ia[0], &ia[12] );

          int search_value = 18;
          int *ptr = lower_bound( ia, ia+12, search_value );

          // печатается:
          // Первый элемент, перед которым можно вставить 18, - это 19
          // Предыдущее значение равно 17

          cout << "Первый элемент, перед которым можно вставить "
               << search_value
               << ", - это "
               << *ptr << endl
               << "Предыдущее значение равно "
               << *(ptr-1) << endl;

          vector< int, allocator > ivec( ia, ia+12 );

          // отсортировать в порядке возрастания ...
          sort( ivec.begin(), ivec.end(), greater<int>() );

          search_value = 26;
          vector< int, allocator >::iterator iter;

          // необходимо указать, как именно
     // осуществлялась сортировка ...

          iter = lower_bound( ivec.begin(), ivec.end(),
                         search_value, greater<int>() );

          // печатается:
          // Первый элемент, перед которым можно вставить 26, - это 26
          // Предыдущее значение равно 29

          cout << "Первый элемент, перед которым можно вставить "
               << search_value
               << ", - это "
               << *iter << endl
               << "Предыдущее значение равно "
               << *(iter-1) << endl;

          return 0;

}
```

Алгоритм max()

```
template< class Type >
const Type&
max( const Type &aval, const Type &bval );

template< class Type, class Compare >
const Type&
max( const Type &aval, const Type &bval, Compare comp );
```

max() возвращает наибольшее из двух значений aval и bval. В первом варианте используется оператор “больше”, определенный в классе Type; во втором – операция сравнения comp.

```
template< class ForwardIterator >
ForwardIterator
max_element( ForwardIterator first,
             ForwardIterator last );

template< class ForwardIterator, class Compare >
ForwardIterator
max_element( ForwardIterator first,
             ForwardIterator last, Compare comp );
```

Алгоритм max_element()

max_element() возвращает итератор, указывающий на элемент, который содержит наибольшее значение в последовательности, ограниченной диапазоном [first,last). В первом варианте используется оператор “больше”, определенный для типа элементов контейнера; во втором – операция сравнения comp.

```
template< class Type >
const Type&
min( const Type &aval, const Type &bval );

template< class Type, class Compare >
const Type&
min( const Type &aval, const Type &bval, Compare comp );
```

Алгоритм min()

min() возвращает меньшее из двух значений aval и bval. В первом варианте используется оператор “меньше”, определенный для типа Type; во втором – операция сравнения comp.

```
template< class ForwardIterator >
ForwardIterator
min_element( ForwardIterator first,
             ForwardIterator last );

template< class ForwardIterator, class Compare >
ForwardIterator
min_element( ForwardIterator first,
```

Алгоритм min_element()

```
             ForwardIterator last, Compare comp );
```

max_element() возвращает итератор, указывающий на элемент, который содержит наименьшее значение последовательности, ограниченной диапазоном [first,last). В первом варианте используется оператор “меньше”, определенный для типа элементов

```
// иллюстрирует max(), min(), max_element(), min_element()

#include <algorithm>
#include <vector>
#include <iostream.h>

int main()
{
    int ia[] = { 7, 5, 2, 4, 3 };
    const vector< int, allocator > ivec( ia, ia+5 );

    int mval = max( max( max( max(ivec[4],ivec[3]),
                                ivec[2]),ivec[1]),ivec[0]);

    // вывод: результат вложенных вызовов max() равен: 7
    cout << "результат вложенных вызовов max() равен: "
         << mval << endl;

    mval = min( min( min( min(ivec[4],ivec[3]),
                            ivec[2]),ivec[1]),ivec[0]);

    // вывод: результат вложенных вызовов min() равен: 2
    cout << "результат вложенных вызовов min() равен: "
         << mval << endl;

    vector< int, allocator >::const_iterator iter;
    iter = max_element( ivec.begin(), ivec.end() );

    // вывод: результат вложенных вызовов max_element() также равен: 7
    cout << "результат вложенных вызовов max_element() также равен: "
         << *iter << endl;

    iter = min_element( ivec.begin(), ivec.end() );

    // вывод: результат вложенных вызовов min_element() также равен: 2
    cout << "результат вложенных вызовов min_element() также равен: "
         << *iter << endl;
```

контейнера; во втором – операция сравнения comp.

```
}
```

```
template< class InputIterator1, class InputIterator2,
          class OutputIterator >
OutputIterator
merge( InputIterator1 first1, InputIterator1 last1,
       InputIterator2 first2, InputIterator2 last2,
       OutputIterator result );

template< class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare >
OutputIterator
merge( InputIterator1 first1, InputIterator1 last1,
       InputIterator2 first2, InputIterator2 last2,
```

Алгоритм merge()

```
       OutputIterator result, Compare comp );
```

`merge()` объединяет две отсортированные последовательности, ограниченные диапазонами `[first1,last1)` и `[first2,last2)`, в единую отсортированную последовательность, начинающуюся с позиции `result`. Результирующий итератор записи указывает на элемент за концом новой последовательности. В первом варианте для упорядочения используется оператор “меньше”, определенный для типа элементов контейнера; во втором – операция сравнения `comp`.

```
#include <algorithm>
#include <vector>
#include <list>
#include <deque>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

void (*pfi)( int ) = print_elements;

int main()
{
          int ia[] =  {29,23,20,22,17,15,26,51,19,12,35,40};
          int ia2[] = {74,16,39,54,21,44,62,10,27,41,65,71};

          vector< int, allocator > vec1( ia,  ia +12 ),
                                   vec2( ia2, ia2+12 );
          int ia_result[24];
          vector< int, allocator > vec_result(vec1.size()+vec2.size());

          sort( ia,  ia +12 );
     sort( ia2, ia2+12 );

     // печатается:
     // 10 12 15 16 17 19 20 21 22 23 26 27 29 35
     //             39 40 41 44 51 54 62 65 71 74

          merge( ia, ia+12, ia2, ia2+12, ia_result );
          for_each( ia_result, ia_result+24, pfi ); cout << "\n\n";

          sort( vec1.begin(), vec1.end(), greater<int>() );
          sort( vec2.begin(), vec2.end(), greater<int>() );

          merge( vec1.begin(), vec1.end(),
            vec2.begin(), vec2.end(),
            vec_result.begin(), greater<int>() );

     // печатается: 74 71 65 62 54 51 44 41 40 39 35 29 27 26 23 22
     //                                    21 20 19 17 16 15 12 10
     for_each( vec_result.begin(), vec_result.end(), pfi );
     cout << "\n\n";

}
```

```
template< class InputIterator1, class InputIterator2 >
pair<InputIterator1, InputIterator2>
mismatch( InputIterator1 first,
          InputIterator1 last, InputIterator2 first2 );

template< class InputIterator1, class InputIterator2,
          class BinaryPredicate >
pair<InputIterator1, InputIterator2>
mismatch( InputIterator1 first, InputIterator1 last,
```

Алгоритм mismatch()

```
          InputIterator2 first2, BinaryPredicate pred );
```

mismatch() сравнивает две последовательности и находит первую позицию, где элементы различны. Возвращается пара итераторов, каждый из которых указывает на эту позицию в соответствующей последовательности. Если все элементы одинаковы, то каждый итератор в паре указывает на элемент last в своем контейнере. Так, если даны последовательности meet и meat, то оба итератора указывают на третий элемент. В первом варианте для сравнения элементов применяется оператор равенства, а во втором – операция сравнения, заданная пользователем. Если вторая последовательность длиннее первой, "лишние" элементы игнорируются; если же она

```
#include <algorithm>
#include <list>
#include <utility>
#include <iostream.h>

class equal_and_odd{
public:
          bool operator()( int ival1, int ival2 )
     {
        // оба значения равны друг другу?
        // оба равны нулю? оба нечетны?

        return ( ival1 == ival2 &&
               ( ival1 == 0 || ival1%2 ));
          }
};

int main()
{
          int ia[] =  { 0,1,1,2,3,5,8,13 };
          int ia2[] = { 0,1,1,2,4,6,10   };

          pair<int*,int*> pair_ia = mismatch( ia, ia+7, ia2 );

     // печатается: первая пара неодинаковых: ia: 3 и ia2: 4
     cout << "первая пара неодинаковых: ia: "
               << *pair_ia.first << " и ia2: "
          << *pair_ia.second << endl;

          list<int,allocator> ilist(  ia,  ia+7  );
          list<int,allocator> ilist2( ia2, ia2+7 );

          typedef list<int,allocator>::iterator iter;
          pair<iter,iter> pair_ilist =
          mismatch( ilist.begin(),  ilist.end(),
                    ilist2.begin(), equal_and_odd() );

     // печатается: первая пара неодинаковых: либо не равны, либо не
          нечетны:
     //             ilist: 2 и ilist2: 2

     cout << "первая пара неодинаковых: либо не равны, "
          << "либо не нечетны: \n\tilist: "
          << *pair_ilist.first << " и ilist2: "
          << *pair_ilist.second << endl;
```

короче, то поведение программы не определено.

```
}
```

```
template < class BidirectionalIterator >
bool
next_permutation( BidirectionalIterator first,
                  BidirectionalIterator last );

template < class BidirectionalIterator, class Compare >
bool
next_permutation( BidirectionalIterator first,
```

Алгоритм next_permutation()

```
                  BidirectionalIterator last, class Compare );
```

`next_permutation()` берет последовательность, ограниченную диапазоном `[first, last)`, и, считая ее перестановкой, возвращает следующую за ней (о том, как упорядочиваются перестановки, говорилось в разделе 12.5). Если следующей перестановки не существует, алгоритм возвращает `false`, иначе – `true`. В первом варианте для определения следующей перестановки используется оператор “меньше” в классе элементов контейнера, а во втором – операция сравнения `comp`. Последовательные обращения к `next_permutation()` генерируют все возможные перестановки только в том случае, когда исходная последовательность отсортирована. Если бы в показанной ниже программе мы предварительно не отсортировали строку `musil`, получив `ilmsu`, то не удалось бы сгенерировать все перестановки.

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

void print_char( char elem ) { cout << elem ; }
void (*ppc)( char ) = print_char;

/* печатается:
ilmsu   ilmus   ilsmu   ilsum   ilums   ilusm   imlsu   imlus
imslu   imsul   imuls   imusl   islmu   islum   ismlu   ismul
isulm   isuml   iulms   iulsm   iumls   iumsl   iuslm   iusml
limsu   limus   lismu   lisum   liums   liusm   lmisu   lmius
lmsiu   lmsui   lmuis   lmusi   lsimu   lsium   lsmiu   lsmui
lsuim   lsumi   luims   luism   lumis   lumsi   lusim   lusmi
milsu   milus   mislu   misul   miuls   miusl   mlisu   mlius
mlsiu   mlsui   mluis   mlusi   msilu   msiul   msliu   mslui
msuil   msuli   muils   muisl   mulis   mulsi   musil   musli
silmu   silum   simlu   simul   siulm   siuml   slimu   slium
slmiu   slmui   sluim   slumi   smilu   smiul   smliu   smlui
smuil   smuli   suilm   suiml   sulim   sulmi   sumil   sumli
uilms   uilsm   uimls   uimsl   uislm   uisml   ulims   ulism
ulmis   ulmsi   ulsim   ulsmi   umils   umisl   umlis   umlsi
umsil   umsli   usilm   usiml   uslim   uslmi   usmil   usmli
*/

int main()
{
          vector<char,allocator> vec(5);

          // последовательность символов: musil
          vec[0] = 'm'; vec[1] = 'u'; vec[2] = 's';
          vec[3] = 'i'; vec[4] = 'l';

          int cnt = 2;
          sort( vec.begin(), vec.end() );
          for_each( vec.begin(), vec.end(), ppc ); cout << "\t";

          // генерируются все перестановки строки "musil"
          while( next_permutation( vec.begin(), vec.end()))
    {
       for_each( vec.begin(), vec.end(), ppc );
       cout << "\t";

       if ( ! ( cnt++ % 8 )) {
            cout << "\n";
           cnt = 1;
       }
         }

         cout << "\n\n";
         return 0;

}
```

```
template < class RandomAccessIterator >
void
nth_element( RandomAccessIterator first,
             RandomAccessIterator nth,
             RandomAccessIterator last );

template < class RandomAccessIterator, class Compare >
void
nth_element( RandomAccessIterator first,
             RandomAccessIterator nth,
             RandomAccessIterator last, Compare comp );
```

Алгоритм nth_element()

`nth_element()` переупорядочивает последовательность, ограниченную диапазоном `[first,last)`, так что все элементы, меньшие чем тот, на который указывает итератор `nth`, оказываются перед ним, а все большие элементы – после. Например, если есть массив

```
int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};
```

то вызов `nth_element()`, в котором `nth` указывает на седьмой элемент (его значение равно 26):

```
nth_element( &ia[0], &ia[6], &ia[2] );
```

генерирует последовательность, в которой семь элементов, меньших 26, оказываются слева от 26, а четыре элемента, больших 26, справа:

```
{23,20,22,17,15,19,12,26,51,35,40,29}
```

При этом не гарантируется, что элементы, расположенные по обе стороны от `nth`, упорядочены. В первом варианте для сравнения используется оператор “меньше”, определенный для типа элементов контейнера, во втором – бинарная операция сравнения, заданная программистом.

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

/* печатается:
исходный вектор: 29 23 20 22 17 15 26 51 19 12 35 40
вектор, отсортированный относительно элемента 26
12 15 17 19 20 22 23 26 51 29 35 40
вектор, отсортированный по убыванию относительно элемента 23
40 35 29 51 26 23 22 20 19 17 15 12
*/

int main()
{
          int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};
          vector< int,allocator > vec( ia, ia+12 );
          ostream_iterator<int> out( cout," " );

          cout << "исходный вектор: ";
          copy( vec.begin(), vec.end(), out ); cout << endl;

          cout << "вектор, отсортированный относительно элемента "
               << *( vec.begin()+6 ) << endl;
          nth_element( vec.begin(), vec.begin()+6, vec.end() );
          copy( vec.begin(), vec.end(), out ); cout << endl;

          cout << " вектор, отсортированный по убыванию "
          << "относительно элемента "
               << *( vec.begin()+6 ) << endl;
          nth_element( vec.begin(), vec.begin()+6,
                       vec.end(),   greater<int>() );
          copy( vec.begin(), vec.end(), out ); cout << endl;

}
```

```
template < class RandomAccessIterator >
void
partial_sort( RandomAccessIterator first,
              RandomAccessIterator middle,
              RandomAccessIterator last );

template < class RandomAccessIterator, class Compare >
void
partial_sort( RandomAccessIterator first,
              RandomAccessIterator middle,
```

Алгоритм partial_sort()

```
              RandomAccessIterator last, Compare comp );
```

`partial_sort()` сортирует часть последовательности, укладывающуюся в диапазон `[first,middle)`. Элементы в диапазоне `[middle,last)` остаются неотсортированными. Например, если дан массив

```
int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};
```

то вызов partial_sort(),где middle указывает на шестой элемент:

```
partial_sort( &ia[0], &ia[5], &ia[12] );
```

генерирует последовательность, в которой наименьшие пять (т.е. middle-first) элементов отсортированы:

```
{12,15,17,19,20,29,23,22,26,51,35,40}.
```

Элементы от middle до last-1 не расположены в каком-то определенном порядке, хотя значения каждого из них лежат вне отсортированной последовательности. В первом варианте для сравнения используется оператор “меньше”, определенный для типа элементов контейнера, а во втором – операция сравнения comp.

```
template < class InputIterator, class RandomAccessIterator >
RandomAccessIterator
partial_sort_copy( InputIterator first, InputIterator last,
             RandomAccessIterator result_first,
             RandomAccessIterator result_last );

template < class InputIterator, class RandomAccessIterator,
           class Compare >
RandomAccessIterator
partial_sort_copy( InputIterator first, InputIterator last,
             RandomAccessIterator result_first,
             RandomAccessIterator result_last,
             Compare comp );
```

Алгоритм partial_sort_copy()

partial_sort_copy() ведет себя так же, как partial_sort(), только частично упорядоченная последовательность копируется в контейнер, ограниченный диапазоном [result_first,result_last] (если мы задаем отдельный контейнер для копирования результата, то в нем оказывается упорядоченная последовательность). Например, даны

```
int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};
```

два массива:

```
int ia2[5];
```

Тогда обращение к partial_sort_copy(), где в качестве middle указан восьмой

```
partial_sort_copy( &ia[0], &ia[7], &ia[12],
```

элемент:

```
                   &ia2[0], &ia2[5] );
```

заполняет массив ia2 пятью отсортированными элементами: {12,15,17,19,20}.

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

/*
* печатается:
  исходный вектор: 69 23 80 42 17 15 26 51 19 12 35 8
  результат применения partial_sort() к вектору: семь элементов
  8 12 15 17 19 23 26 80 69 51 42 35
  результат применения partial_sort_copy() к первым семи
  элементам вектора в порядке убывания
  26 23 19 17 15 12 8
*/

int main()
{
          int ia[] = { 69,23,80,42,17,15,26,51,19,12,35,8 };
          vector< int,allocator > vec( ia, ia+12 );
          ostream_iterator<int> out( cout," " );

          cout << "исходный вектор: ";
          copy( vec.begin(), vec.end(), out ); cout << endl;

     cout << "результат применения partial_sort() к вектору: "
          << "семь элементов \n";
     partial_sort( vec.begin(), vec.begin()+7, vec.end() );
     copy( vec.begin(), vec.end(), out ); cout << endl;

          vector< int, allocator > res(7);
     cout << " результат применения partial_sort_copy() к первым семи
          \n\t"
               << "элементам вектора в порядке убывания \n";

     partial_sort_copy( vec.begin(), vec.begin()+7, res.begin(),
                        res.end(), greater<int>() );
     copy( res.begin(), res.end(), out ); cout << endl;
```

Оставшиеся два элемента отсортированы не будут.

```
}
```

```
template < class InputIterator, class OutputIterator >
OutputIterator
partial_sum(
     InputIterator first, InputIterator last,
     OutputIterator result );

template < class InputIterator, class OutputIterator,
           class BinaryOperation >
OutputIterator
partial_sum(
     InputIterator first, InputIterator last,
```

Алгоритм partial_sum()

```
    OutputIterator result, BinaryOperation op );
```

Первый вариант partial_sum() создает из последовательности, ограниченной диапазоном [first,last), новую последовательность, в которой значение каждого элемента равно сумме всех предыдущих, включая и данный. Так, из последовательности {0,1,1,2,3,5,8} будет создана {0,1,2,4,7,12,20}, где, например, четвертый элемент равен сумме трех предыдущих (0,1,1) и его самого (2), что дает значение 4.

Во втором варианте вместо оператора сложения используется бинарная операция, заданная программистом. Предположим, мы задали последовательность {1,2,3,4} и объект-функцию times<int>. Результатом будет {1,2,6,24}. В обоих случаях итератор записи OutputIterator указывает на элемент за последним элементом новой последовательности.

partial_sum() – это один из численных алгоритмов. Для его использования

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

/*
 * печатается:
   элементы: 1 3 4 5 7 8 9
   частичная сумма элементов:
   1 4 8 13 20 28 37
   частичная сумма элементов с использованием times<int>():
   1 3 12 60 420 3360 30240
*/

int main()
{
          const int ia_size = 7;
          int ia[ ia_size ] = { 1, 3, 4, 5, 7, 8, 9 };
          int ia_res[ ia_size ];

          ostream_iterator< int  > outfile( cout, " "  );
          vector< int, allocator > vec( ia, ia+ia_size );
          vector< int, allocator > vec_res( vec.size() );

          cout << "элементы: ";
     copy( ia, ia+ia_size, outfile ); cout << endl;

          cout << "частичная сумма элементов:\n";
          partial_sum( ia, ia+ia_size, ia_res );
          copy( ia_res, ia_res+ia_size, outfile ); cout << endl;

          cout << "частичная сумма элементов с использованием
          times<int>():\n";
          partial_sum( vec.begin(), vec.end(), vec_res.begin(),
                  times<int>() );

          copy( vec_res.begin(), vec_res.end(), outfile );
     cout << endl;
```

необходимо включить в программу стандартный заголовочный файл <numeric>.

```
}
```

```
template < class BidirectionalIterator, class UnaryPredicate >
BidirectionalIterator
partition(
     BidirectionalIterator first,
```

Алгоритм partition()

```
     BidirectionalIterator last, UnaryPredicate pred );
```

`partition()` переупорядочивает элементы в диапазоне `[first,last)`. Все элементы, для которых предикат `pred` равен `true`, помещаются перед элементами, для которых он равен `false`. Например, если дана последовательность {0,1,2,3,4,5,6} и предикат, проверяющий целое число на четность, то мы получим две последовательности – {0,2,4,6} и {1,3,5}. Хотя гарантируется, что четные элементы будут помещены перед нечетными, их первоначальное взаимное расположение может и не сохраниться, т.е. 4 может оказаться перед 2, а 5 перед 1. Сохранение относительного порядка обеспечивает алгоритм `stable_partition()`, рассматриваемый ниже.

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

class even_elem {
public:
    bool operator()( int elem )
         { return elem%2 ? false : true; }
};

/*
 * печатается:
   исходная последовательность:
   29 23 20 22 17 15 26 51 19 12 35 40
   разбиение, основанное на четности элементов:
   40 12 20 22 26 15 17 51 19 23 35 29
   разбиение, основанное на сравнении с 25:
   12 23 20 22 17 15 19 51 26 29 35 40
*/

int main()
{
          const int ia_size = 12;
          int ia[ia_size]   = { 29,23,20,22,17,15,26,51,19,12,35,40 };

          vector< int, allocator > vec( ia, ia+ia_size );
          ostream_iterator< int >  outfile( cout, " "  );

          cout << "исходная последовательность: \n";
          copy( vec.begin(), vec.end(), outfile ); cout << endl;

          cout << "разбиение, основанное на четности элементов:\n";
          partition( &ia[0], &ia[ia_size], even_elem() );
          copy( ia, ia+ia_size, outfile ); cout << endl;

          cout << "разбиение, основанное на сравнении с 25:\n";
          partition( vec.begin(), vec.end(), bind2nd(less<int>(),25)  );
          copy( vec.begin(), vec.end(), outfile ); cout << endl;

}
```

```
template < class BidirectionalIterator >
bool
prev_permutation( BidirectionalIterator first,
                  BidirectionalIterator last );

template < class BidirectionalIterator, class Compare >
bool
prev_permutation( BidirectionalIterator first,
                  BidirectionalIterator last, class Compare );
```

Алгоритм prev_permutation()

`prev_permutation()` берет последовательность, ограниченную диапазоном `[first, last)`, и, рассматривая ее как перестановку, возвращает предшествующую ей

(о том, как упорядочиваются перестановки, говорилось в разделе 12.5). Если предыдущей перестановки не существует, алгоритм возвращает false, иначе true. В первом варианте для определения предыдущей перестановки используется оператор "меньше" для типа элементов контейнера, а во втором – бинарная операция сравнения,

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

// печатается:    n d a   n a d   d n a   d a n   a n d   a d n

int main()
{
          vector< char, allocator > vec( 3 );
          ostream_iterator< char > out_stream( cout, " " );

          vec[0] = 'n'; vec[1] = 'd'; vec[2] = 'a';
          copy( vec.begin(), vec.end(), out_stream ); cout << "\t";

          // сгенерировать все перестановки "dan"
          while( prev_permutation( vec.begin(), vec.end() )) {
                 copy( vec.begin(), vec.end(), out_stream );
                 cout << "\t";
          }

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

заданная программистом.

```
}
```

```
template < class RandomAccessIterator >
void
random_shuffle( RandomAccessIterator first,
                RandomAccessIterator last );

template < class RandomAccessIterator,
           class RandomNumberGenerator >
void
random_shuffle( RandomAccessIterator first,
                RandomAccessIterator last,
```

Алгоритм random_shuffle()

```
                RandomNumberGenerator rand);
```

random_shuffle() переставляет элементы из диапазона [first,last) в случайном порядке. Во втором варианте можно передать объект-функцию или указатель на функцию, генерирующую случайные числа. Ожидается, что генератор rand возвращает значение типа double в интервале [0,1].

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

int main()
{
          vector< int, allocator > vec;
          for ( int ix = 0; ix < 20; ix++ )
                vec.push_back( ix );

          random_shuffle( vec.begin(), vec.end() );

          // печатает:
          // random_shuffle для последовательности 1 .. 20:
          // 6 11 9 2 18 12 17 7 0 15 4 8 10 5 1 19 13 3 14 16
          cout << "random_shuffle для последовательности 1 .. 20:\n";
          copy( vec.begin(), vec.end(), ostream_iterator< int >( cout,"
          " ));
}
```

```
template< class ForwardIterator, class Type >
ForwardIterator
remove( ForwardIterator first,
```

Алгоритм remove()

```
        ForwardIterator last, const Type &value );
```

`remove()` удаляет из диапазона `[first,last)` все элементы со значением `value`. Этот алгоритм (как и `remove_if()`) на самом деле не исключает элементы из контейнера (т.е. размер контейнера сохраняется), а перемещает каждый оставляемый элемент в очередную позицию, начиная с `first`. Возвращаемый итератор указывает на элемент, следующий за позицией, в которую помещен последний неудаленный элемент. Рассмотрим, например, последовательность {0,1,0,2,0,3,0,4}. Предположим, что нужно удалить все нули. В результате получится последовательность {1,2,3,4,0,4,0,4}. 1 помещена в первую позицию, 2 – во вторую, 3 – в третью и 4 – в четвертую. Элементы, начиная с 0 в пятой позиции, – это “отходы” алгоритма. Возвращенный итератор указывает на 0 в пятой позиции. Обычно этот итератор затем передается алгоритму `erase()`, который удаляет неподходящие элементы. (При работе со встроенным массивом лучше использовать алгоритмы `remove_copy()` и `remove_copy_if()`, а не `remove()` и `remove_if()`, поскольку его размер невозможно изменить)

Алгоритм remove_copy()

```
template< class InputIterator, class OutputIterator,
          class Type >
OutputIterator
remove_copy( InputIterator first, InputIterator last,
             OutputIterator result, const Type &value );
```

remove_copy() копирует все элементы, кроме имеющих значение value, в контейнер, на начало которого указывает result. Возвращаемый итератор указывает на элемент за последним скопированным. Исходный контейнер не изменяется.

```
#include <algorithm>
#include <vector>
#include <assert.h>
#include <iostream.h>

/* печатается:
   исходный вектор:
   0 1 0 2 0 3 0 4 0 5
   вектор после remove до erase():
   1 2 3 4 5 3 0 4 0 5
   вектор после erase():
   1 2 3 4 5
   массив после remove_copy()
   1 2 3 4 5
*/

int main()
{
          int value = 0;
          int ia[] = { 0, 1, 0, 2, 0, 3, 0, 4, 0, 5 };

          vector< int, allocator >  vec( ia, ia+10 );
          ostream_iterator< int >   ofile( cout," ");
          vector< int, allocator >::iterator vec_iter;

          cout << "исходный вектор:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

          vec_iter = remove( vec.begin(), vec.end(), value );

          cout << "вектор после remove до erase():\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

          // удалить из контейнера неподходящие элементы
          vec.erase( vec_iter, vec.end() );

          cout << "вектор после erase():\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

     int ia2[5];
     vector< int, allocator > vec2( ia, ia+10 );
     remove_copy( vec2.begin(), vec2.end(), ia2, value );

     cout << "массив после remove_copy():\n";
     copy( ia2, ia2+5, ofile ); cout << endl;
}
```

```
template< class ForwardIterator, class Predicate >
ForwardIterator
remove_if( ForwardIterator first,
```

Алгоритм remove_if()

```
           ForwardIterator last, Predicate pred );
```

`remove_if()` удаляет из диапазона `[first, last)` все элементы, для которых значение предиката `pred` равно `true`. `remove_if()` (как и `remove()`) фактически не исключает удаленные элементы из контейнера. Вместо этого каждый оставляемый элемент перемещается в очередную позицию, начиная с `first`. Возвращаемый итератор указывает на элемент, следующий за позицией, в которую помещен последний неудаленный элемент. Обычно этот итератор затем передается алгоритму `erase()`, который удаляет неподходящие элементы. (Для встроенных массивов лучше использовать алгоритм `remove_copy_if()`.)

```
template< class InputIterator, class OutputIterator,
          class Predicate >
OutputIterator
remove_copy_if( InputIterator first, InputIterator last,
```

Алгоритм remove_copy_if()

```
                OutputIterator result, Predicate pred );
```

`remove_copy_if()` копирует все элементы, для которых предикат `pred` равен `false`, в контейнер, на начало которого указывает итератор `result`. Возвращаемый итератор указывает на элемент, расположенный за последним скопированным. Исходный контейнер остается без изменения.

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

/* печатается:
   исходная последовательность:
   0 1 1 2 3 5 8 13 21 34
   последовательность после применения remove_if < 10:
   13 21 34
   последовательность после применения remove_copy_if четное:
   1 1 3 5 13 21
*/

class EvenValue {
public:
          bool operator()( int value ) {
          return value % 2 ? false : true; }
};

int main()
{
          int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };

          vector< int, allocator >::iterator iter;
          vector< int, allocator >  vec( ia, ia+10 );
     ostream_iterator< int >   ofile( cout, " " );

     cout << "исходная последовательность:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

          iter = remove_if( vec.begin(), vec.end(),
                      bind2nd(less<int>(),10) );
          vec.erase( iter, vec.end() );

     cout << "последовательность после применения remove_if < 10:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

          vector< int, allocator > vec_res( 10 );
          iter = remove_copy_if( ia, ia+10, vec_res.begin(), EvenValue()
          );

     cout << "последовательность после применения remove_copy_if
          четное:\n";
     copy( vec_res.begin(), iter, ofile ); cout << '\n';

}
```

Алгоритм replace()

```
template< class ForwardIterator, class Type >
void
replace( ForwardIterator first, ForwardIterator last,
         const Type& old_value, const Type& new_value );
```

replace() заменяет в диапазоне [first,last) все элементы со значением old_value на new_value.

```
template< class InputIterator, class InputIterator,
          class Type >
OutputIterator
replace_copy( InputIterator first, InputIterator last,
              class OutputIterator result,
```

Алгоритм replace_copy()

```
              const Type& old_value, const Type& new_value );
```

replace_copy() ведет себя так же, как replace(), только новая последовательность копируется в контейнер, начиная с result. Возвращаемый итератор указывает на элемент, расположенный за последним скопированным. Исходный контейнер остается без изменения.

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

/* печатается:
   исходная последовательность:
   Christopher Robin Mr. Winnie the Pooh Piglet Tigger Eeyore
   последовательность после применения replace():
   Christopher Robin Pooh Piglet Tigger Eeyore
*/

int main()
{
          string oldval( "Mr. Winnie the Pooh" );
          string newval( "Pooh" );

     ostream_iterator< string >  ofile( cout, " " );
          string sa[] = {
          "Christopher Robin", "Mr. Winnie the Pooh",
          "Piglet", "Tigger", "Eeyore"
     };

          vector< string, allocator > vec( sa, sa+5 );
     cout << "исходная последовательность:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

          replace( vec.begin(), vec.end(), oldval, newval );

     cout << "последовательность после применения replace():\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

          vector< string, allocator > vec2;
          replace_copy( vec.begin(), vec.end(),
                   inserter( vec2, vec2.begin() ),
                   newval, oldval );

     cout << "последовательность после применения replace_copy():\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

}
```

```
template< class ForwardIterator, class Predicate, class Type >
void
replace_if( ForwardIterator first, ForwardIterator last,
```

Алгоритм replace_if()

```
            Predicate pred, const Type& new_value );
```

replace_if() заменяет значения всех элементов в диапазоне [first,last), для которых предикат pred равен true, на new_value.

```
template< class ForwardIterator, class OutputIterator,
          class Predicate, class Type >
OutputIterator
replace_copy_if( ForwardIterator first, ForwardIterator last,
                 class OutputIterator result,
```

Алгоритм replace_copy_if()

```
                 Predicate pred, const Type& new_value );
```

`replace_copy_if()` ведет себя так же, как `replace_if()`, только новая последовательность копируется в контейнер, начиная с `result`. Возвращаемый итератор указывает на элемент, расположенный за последним скопированным. Исходный контейнер остается без изменения.

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

/*
   исходная последовательность:
   0 1 1 2 3 5 8 13 21 34
   последовательность после применения replace_if < 10 с заменой на 0:
   0 0 0 0 0 0 0 13 21 34
   последовательность после применения replace_if четное с заменой на 0:
   0 1 1 0 3 5 0 13 21 0
*/

class EvenValue {
public:
          bool operator()( int value ) {
          return value % 2 ? false : true; }
};

int main()
{
          int new_value = 0;

          int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
          vector< int, allocator > vec( ia, ia+10 );
     ostream_iterator< int >  ofile( cout, " " );

     cout << "исходная последовательность:\n";
     copy( ia, ia+10, ofile ); cout << '\n';

          replace_if( &ia[0], &ia[10],
                bind2nd(less<int>(),10), new_value );

     cout << "последовательность после применения replace_if < 10 "
          << "с заменой на 0:\n";
     copy( ia, ia+10, ofile ); cout << '\n';

          replace_if( vec.begin(), vec.end(),
                EvenValue(), new_value );

     cout << "последовательность после применения replace_if четное"
          << "с заменой на 0:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

}
```

```
template< class BidirectionalIterator >
void
reverse( BidirectionalIterator first,
```

Алгоритм reverse()

```
         BidirectionalIterator last );
```

`reverse()` меняет порядок элементов контейнера в диапазоне `[first,last)` на противоположный. Например, если есть последовательность {0,1,1,2,3}, то после обращения получится {3,2,1,1,0}.

```
template< class BidirectionalIterator, class OutputIterator >
OutputIterator
reverse_copy( BidirectionalIterator first,
```

Алгоритм reverse_copy()

```
              BidirectionalIterator last, OutputIterator result );
```

`reverse_copy()` ведет себя так же, как `reverse()`, только новая последовательность копируется в контейнер, начиная с `result`. Возвращаемый итератор указывает на элемент, расположенный за последним скопированным. Исходный контейнер остается без изменения.

```
#include <algorithm>
#include <list>
#include <string>
#include <iostream.h>

/* печатается:
   Исходная последовательность строк:
        Signature of all things I am here to
        read seaspawn and seawrack that rusty boot

   Последовательность строк после применения reverse():
        boot rusty that seawrack and seaspawn read to
        here am I things all of Signature
*/

class print_elements {
public:
          void operator()( string elem ) {
          cout << elem
               << ( _line_cnt++%8 ? " " : "\n\t" );
          }

          static void reset_line_cnt() { _line_cnt = 1; }

private:
          static int _line_cnt;
};

int print_elements::_line_cnt = 1;

int main()
{
          string sa[] = { "Signature", "of", "all", "things",
                 "I", "am", "here", "to", "read",
                 "seaspawn", "and", "seawrack", "that",
                 "rusty", "boot"
          };

          list< string, allocator > slist( sa, sa+15 );

          cout << "Исходная последовательность строк:\n\t";
        for_each( slist.begin(), slist.end(), print_elements() );
          cout << "\n\n";

          reverse( slist.begin(), slist.end() );

          print_elements::reset_line_cnt();

          cout << "Последовательность строк после применения
          reverse():\n\t";
     for_each( slist.begin(), slist.end(), print_elements() ); cout <<
          "\n";

          list< string, allocator > slist_copy( slist.size() );
          reverse_copy( slist.begin(), slist.end(),
                   slist_copy.begin() );

}
```

```
template< class ForwardIterator >
void
rotate( ForwardIterator first,
```

Алгоритм rotate()

```
        ForwardIterator middle, ForwardIterator last );
```

`rotate()` перемещает элементы из диапазона `[first,last)` в конец контейнера. Элемент, на который указывает `middle`, становится первым. Например, для слова "`hissboo`" вращение вокруг буквы '`b`' превращает слово в "`boohiss`".

```
template< class ForwardIterator, class OutputIterator >
OutputIterator
rotate_copy( ForwardIterator first, ForwardIterator middle,
```

Алгоритм rotate_copy()

```
             ForwardIterator last, OutputIterator result );
```

`rotate_copy()` ведет себя так же, как `rotate()`, только новая последовательность копируется в контейнер, начиная с `result`. Возвращаемый итератор указывает на элемент, расположенный за последним скопированным. Исходный контейнер остается без изменения.

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

/* печатается:
   исходная последовательность:
   1 3 5 7 9 0 2 4 6 8 10
   вращение вокруг среднего элемента(0) ::
   0 2 4 6 8 10 1 3 5 7 9
   вращение вокруг предпоследнего элемента(8) ::
   8 10 1 3 5 7 9 0 2 4 6
   rotate_copy вокруг среднего элемента ::
   7 9 0 2 4 6 8 10 1 3 5
*/
int main()
{
          int ia[] = { 1, 3, 5, 7, 9, 0, 2, 4, 6, 8, 10 };

          vector< int, allocator > vec( ia, ia+11 );
        ostream_iterator< int >  ofile( cout, " " );

     cout << "исходная последовательность:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

          rotate( &ia[0], &ia[5], &ia[11] );

     cout << "вращение вокруг среднего элемента(0) ::\n";
     copy( ia, ia+11, ofile ); cout << '\n';

          rotate( vec.begin(), vec.end()-2, vec.end() );

          cout << "вращение вокруг предпоследнего элемента(8) ::\n";
          copy( vec.begin(), vec.end(), ofile ); cout << '\n';

          vector< int, allocator > vec_res( vec.size() );

     rotate_copy( vec.begin(), vec.begin()+vec.size()/2,
                  vec.end(), vec_res.begin() );

     cout << "rotate_copy вокруг среднего элемента ::\n";
     copy( vec_res.begin(), vec_res.end(), ofile );
     cout << '\n';

}
```

Алгоритм search()

```
template< class ForwardIterator1, class ForwardIterator2 >
ForwardIterator
search( ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2 );

template< class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate >
ForwardIterator
search( ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,

        BinaryPredicate pred );
```

Если даны два диапазона, то `search()` возвращает итератор, указывающий на первую позицию в диапазоне `[first1,last1)`, начиная с которой второй диапазон входит как подпоследовательность. Если подпоследовательность не найдена, возвращается `last1`. Например, в слове `Mississippi` подпоследовательность `iss` встречается дважды, и `search()` возвращает итератор, указывающий на начало первого вхождения. В первом варианте для сравнения элементов используется оператор равенства, во втором –

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

/* печатается:
   Ожидаем найти подстроку 'ate': a t e
   Ожидаем найти подстроку 'vat': v a t
*/

int main()
{
     ostream_iterator< char >  ofile( cout, " " );

          char str[ 25 ]   = "a fine and private place";
          char substr[] = "ate";

          char *found_str = search(str,str+25,substr,substr+3);

          cout << "Ожидаем найти подстроку 'ate': ";
     copy( found_str, found_str+3, ofile ); cout << '\n';

          vector< char, allocator > vec( str, str+24 );
          vector< char, allocator > subvec(3);

          subvec[0]='v'; subvec[1]='a'; subvec[2]='t';

          vector< char, allocator >::iterator iter;
     iter = search( vec.begin(), vec.end(),
                    subvec.begin(), subvec.end(),
                    equal_to< char >() );

          cout << "Ожидаем найти подстроку 'vat': ";
     copy( iter, iter+3, ofile ); cout << '\n';
```

указанная программистом операция сравнения.

```
}
```

```
template< class ForwardIterator, class Size, class Type >
ForwardIterator
search_n( ForwardIterator first, ForwardIterator last,
          Size count, const Type &value );

template< class ForwardIterator, class Size,
          class Type, class BinaryPredicate >
ForwardIterator
search_n( ForwardIterator first, ForwardIterator last,
```

Алгоритм search_n()

```
          Size count, const Type &value, BinaryPredicate pred );
```

search_n() ищет в последовательности [first,last) подпоследовательность, состоящую из count повторений значения value. Если она не найдена, возвращается last. Например, для поиска подстроки ss в строке Mississippi следует задать value равным 's', а count равным 2. Если же нужно найти две расположенные подряд подстроки ssi, то value задается равным "ssi", а count снова 2. search_n() возвращает итератор на первый элемент со значением value. В первом варианте для сравнения элементов используется оператор равенства, во втором – указанная программистом операция сравнения.

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

/* печатается:
   Ожидаем найти два вхождения 'o': o o
   Ожидаем найти подстроку 'mou':  m o u
*/

int main()
{
     ostream_iterator< char >  ofile( cout, " " );

          const char blank = ' ';
          const char oh    = 'o';

          char str[ 26 ]  = "oh my a mouse ate a moose";
          char *found_str = search_n( str, str+25, 2, oh );

          cout << "Ожидаем найти два вхождения 'o': ";
     copy( found_str, found_str+2, ofile ); cout << '\n';

          vector< char, allocator > vec( str, str+25 );

          // найти первую последовательность из трех символов,
          // ни один из которых не равен пробелу: mou of mouse

          vector< char, allocator >::iterator iter;
     iter = search_n( vec.begin(), vec.end(), 3,
                           blank, not_equal_to< char >() );

          cout << "Ожидаем найти подстроку 'mou':  ";
     copy( iter, iter+3, ofile ); cout << '\n';

}
```

```
template< class InputIterator1, class InputIterator2,
          class OutputIterator >
OutputIterator
set_difference( InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2,
                OutputIterator result );

template< class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare >
OutputIterator
set_difference( InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2,
```

Алгоритм set_difference()

```
                OutputIterator result, Compare comp );
```

`set_difference()` строит отсортированную последовательность из элементов, имеющихся в первой последовательности `[first1,last1)`, но отсутствующих во второй – `[first2,last2)`. Например, разность последовательностей {0,1,2,3} и

{0,2,4,6} равна {1,3}. Возвращаемый итератор указывает на элемент за последним помещенным в выходной контейнер result. В первом варианте предполагается, что обе последовательности были отсортированы с помощью оператора “меньше”, определенного для типа элементов контейнера; во втором для упорядочения используется указанная программистом операция comp.

```
template< class InputIterator1, class InputIterator2,
          class OutputIterator >
OutputIterator
set_intersection( InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2,
                  OutputIterator result );

template< class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare >
OutputIterator
set_intersection( InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2,
```

Алгоритм set_intersection()

```
                  OutputIterator result, Compare comp );
```

set_intersection() строит отсортированную последовательность из элементов, встречающихся в обеих последовательностях – [first1,last1) и [first2,last2). Например, пересечение последовательностей {0,1,2,3} и {0,2,4,6} равно {0,2}. Возвращаемый итератор указывает на элемент за последним помещенным в выходной контейнер result. В первом варианте предполагается, что обе последовательности были отсортированы с помощью оператора “меньше”, определенного для типа элементов контейнера; во втором для упорядочения используется указанная программистом операция comp.

```
template< class InputIterator1, class InputIterator2,
          class OutputIterator >
OutputIterator
set_symmetric_difference(
       InputIterator1 first1, InputIterator1 last1,
       InputIterator2 first2, InputIterator2 last2,
       OutputIterator result );

template< class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare >
OutputIterator
set_symmetric_difference(
       InputIterator1 first1, InputIterator1 last1,
       InputIterator2 first2, InputIterator2 last2,
```

Алгоритм set_symmetric_difference()

```
       OutputIterator result, Compare comp );
```

set_symmetric_difference() строит отсортированную последовательность из элементов, которые встречаются только в первой последовательности [first1,last1) или только во второй – [first2,last2). Например, симметрическая разность последовательностей {0,1,2,3} и {0,2,4,6} равна {1,3,4,6}. Возвращаемый итератор указывает на элемент за последним помещенным в выходной контейнер result. В первом варианте предполагается, что обе последовательности были отсортированы с помощью оператора “меньше”, определенного для типа элементов контейнера; во втором для упорядочения используется указанная программистом операция comp.

```
template< class InputIterator1, class InputIterator2,
          class OutputIterator >
OutputIterator
set_union(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result );

template< class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare >
OutputIterator
set_union(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
```

Алгоритм set_union()

```
          OutputIterator result, Compare comp );
```

set_union() строит отсортированную последовательность из элементов, которые встречаются либо в первой последовательности [first1,last1), либо во второй – [first2,last2), либо в обеих. Например, объединение последовательностей {0,1,2,3} и {0,2,4,6} равно {0,1,2,3,4,6}. Если элемент присутствует в обеих последовательностях, то копируется экземпляр из первой. Возвращаемый итератор указывает на элемент за последним помещенным в выходной контейнер result. В первом варианте предполагается, что обе последовательности были отсортированы с помощью оператора “меньше”, определенного для типа элементов контейнера; во втором для упорядочения используется указанная программистом операция comp.

```
#include <algorithm>
#include <set>
#include <string>
#include <iostream.h>

/* печатается:
   элементы множества #1:
        Иа-Иа Пух Пятачок Тигра

   элементы множества #2:
        Бука Пух Слонопотам

   элементы set_union():
        Бука Иа-Иа Пух Пятачок Слонопотам Тигра

   элементы set_intersection():
        Пух

   элементы set_difference():
        Иа-Иа Пятачок Тигра

   элементы_symmetric_difference():
       Бука Иа-Иа Пятачок Слонопотам Тигра
*/

int main()
{
          string str1[] = { "Пух", "Пятачок", "Тигра", "Иа-Иа" };
          string str2[] = { "Пух", "Слонопотам", "Бука" };
     ostream_iterator< string >  ofile( cout, " " );

          set<string,less<string>,allocator> set1( str1, str1+4 );
          set<string,less<string>,allocator> set2( str2, str2+3 );

     cout << "элементы множества #1:\n\t";
     copy( set1.begin(), set1.end(), ofile ); cout << "\n\n";
     cout << "элементы множества #2:\n\t";
     copy( set2.begin(), set2.end(), ofile ); cout << "\n\n";

          set<string,less<string>,allocator> res;
          set_union( set1.begin(), set1.end(),
                set2.begin(), set2.end(),
                inserter( res, res.begin() ));

     cout << "элементы set_union():\n\t";
     copy( res.begin(), res.end(), ofile ); cout << "\n\n";

          res.clear();
          set_intersection( set1.begin(), set1.end(),
                      set2.begin(), set2.end(),
                      inserter( res, res.begin() ));

     cout << "элементы set_intersection():\n\t";
     copy( res.begin(), res.end(), ofile ); cout << "\n\n";

     res.clear();
     set_difference( set1.begin(), set1.end(),
                    set2.begin(), set2.end(),
                    inserter( res, res.begin() ));

     cout << "элементы set_difference():\n\t";
     copy( res.begin(), res.end(), ofile ); cout << "\n\n";

     res.clear();
     set_symmetric_difference( set1.begin(), set1.end(),
                               set2.begin(), set2.end(),
                               inserter( res, res.begin() ));

     cout << "элементы set_symmetric_difference():\n\t";
     copy( res.begin(), res.end(), ofile ); cout << "\n\n";
```

```
}
```

```
template< class RandomAccessIterator >
void
sort( RandomAccessIterator first,
      RandomAccessIterator last );

template< class RandomAccessIterator, class Compare >
void
sort( RandomAccessIterator first,
```

Алгоритм sort()

```
      RandomAccessIterator last, Compare comp );
```

`sort()` переупорядочивает элементы в диапазоне `[first,last)` по возрастанию, используя оператор "меньше", определенный для типа элементов контейнера. Во втором варианте порядок устанавливается операцией сравнения `comp`. (Для сохранения относительного порядка равных элементов пользуйтесь алгоритмом `stable_sort()`.) Мы не приводим пример, специально иллюстрирующий применение алгоритма `sort()`, поскольку его можно найти во многих других программах, в частности в `binary_search()`, `equal_range()` и `inplace_merge()`.

```
template< class BidirectionalIterator, class Predicate >
BidirectionalIterator
stable_partition( BidirectionalIterator first,
                  BidirectionalIterator last,
```

Алгоритм stable_partition()

```
                  Predicate pred );
```

`stable_partition()` ведет себя так же, как `partition()`, но гарантированно сохраняет относительный порядок элементов контейнера. Вот та же программа, что и для алгоритма `partition()`, но с использованием `stable_partition()`.

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

/* печатается:
   исходная последовательность:
   29 23 20 22 17 15 26 51 19 12 35 40
   устойчивое разбиение по четным элементам:
   20 22 26 12 40 29 23 17 15 51 19
   устойчивое разбиение по элементам, меньшим 25:
   23 20 22 17 15 19 12 29 26 51 35 40
*/

class even_elem {
public:
          bool operator()( int elem ) {
          return elem%2 ? false : true;
          }
};

int main()
{
          int ia[] = { 29,23,20,22,17,15,26,51,19,12,35,40 };
          vector< int, allocator > vec( ia, ia+12 );
     ostream_iterator< int >  ofile( cout, " " );

     cout << "исходная последовательность:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

          stable_partition( &ia[0], &ia[12], even_elem() );

     cout << "устойчивое разбиение по четным элементам:\n";
     copy( ia, ia+11, ofile ); cout << '\n';

          stable_partition( vec.begin(), vec.end(),
                       bind2nd(less<int>(),25)  );

     cout << "устойчивое разбиение по элементам, меньшим 25:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

}
```

```
template< class RandomAccessIterator >
void
stable_sort( RandomAccessIterator first,
      RandomAccessIterator last );

template< class RandomAccessIterator, class Compare >
void
stable_sort( RandomAccessIterator first,
```

Алгоритм stable_sort()

```
      RandomAccessIterator last, Compare comp );
```

stable_sort() ведет себя так же, как sort(), но гарантированно сохраняет относительный порядок равных элементов контейнера. Второй вариант упорядочивает

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

/* печатается:
   исходная последовательность:
   29 23 20 22 12 17 15 26 51 19 12 23 35 40
   устойчивая сортировка - по умолчанию в порядке возрастания:
   12 12 15 17 19 20 22 23 23 26 29 35 40 51
   устойчивая сортировка: в порядке убывания:
   51 40 35 29 26 23 23 22 20 19 17 15 12 12
*/

int main()
{
          int ia[] = { 29,23,20,22,12,17,15,26,51,19,12,23,35,40 };
          vector< int, allocator > vec( ia, ia+14 );
     ostream_iterator< int >  ofile( cout, " " );

     cout << "исходная последовательность:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

          stable_sort( &ia[0], &ia[14] );

     cout << "устойчивая сортировка - по умолчанию "
          << "в порядке возрастания:\n";
     copy( ia, ia+14, ofile ); cout << '\n';

          stable_sort( vec.begin(), vec.end(), greater<int>() );

          cout << "устойчивая сортировка: в порядке убывания:\n";
          copy( vec.begin(), vec.end(), ofile ); cout << '\n';
```

элементы на основе заданной программистом операции сравнения comp.

```
}
```

```
template< class Type >
void
```

Алгоритм swap()

```
swap ( Type &ob1, Type &ob2 );
```

swap() обменивает значения объектов ob1 и ob2.

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

/* печатается:
   исходная последовательность:
   3 4 5 0 1 2
   после применения swap() в процедуре пузырьковой сортировки:
   0 1 2 3 4 5
*/

int main()
{
          int ia[]  = { 3, 4, 5, 0, 1, 2 };
          vector< int, allocator > vec( ia, ia+6 );

          for ( int ix = 0; ix < 6; ++ix )
          for ( int iy = ix; iy < 6; ++iy ) {
                if ( vec[iy] < vec[ ix ] )
                     swap( vec[iy], vec[ix] );
          }

          ostream_iterator< int >  ofile( cout, " " );

     cout << "исходная последовательность:\n";
     copy( ia, ia+6, ofile ); cout << '\n';

     cout << "после применения swap() в процедуре "
          << "пузырьковой сортировки:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

}
```

```
template< class ForwardIterator1, class ForwardIterator2 >
ForwardIterator2
swap_ranges( ForwardIterator1 first1, ForwardIterator1 last,
```

Алгоритм swap_ranges()

```
             ForwardIterator2 first2 );
```

swap_ranges() обменивает элементы из диапазона [first1,last) с элементами другого диапазона, начиная с first2. Эти последовательности могут находиться в одном контейнере или в разных. Поведение программы не определено, если они находятся в одном контейнере и при этом частично перекрываются, а также в случае, когда вторая последовательность короче первой. Алгоритм возвращает итератор, указывающий на элемент за последним переставленным.

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

/* печатается:
   исходная последовательность элементов первого контейнера:
   0 1 2 3 4 5 6 7 8 9
   исходная последовательность элементов второго контейнера:
   5 6 7 8 9
   массив после перестановки двух половин:
   5 6 7 8 9 0 1 2 3 4
   первый контейнер после перестановки двух векторов:
   5 6 7 8 9 5 6 7 8 9
   второй контейнер после перестановки двух векторов:
   0 1 2 3 4
*/
int main()
{
          int ia[]  = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
          int ia2[] = { 5, 6, 7, 8, 9 };

          vector< int, allocator > vec( ia, ia+10 );
          vector< int, allocator > vec2( ia2, ia2+5 );

          ostream_iterator< int >  ofile( cout, " " );

     cout << "исходная последовательность элементов первого
          контейнера:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

          cout << "исходная последовательность элементов второго
          контейнера:\n";
     copy( vec2.begin(), vec2.end(), ofile ); cout << '\n';

          // перестановка внутри одного контейнера
          swap_ranges( &ia[0], &ia[5], &ia[5] );

     cout << "массив после перестановки двух половин:\n";
     copy( ia, ia+10, ofile ); cout << '\n';

          // перестановка разных контейнеров
          vector< int, allocator >::iterator last =
     find( vec.begin(), vec.end(), 5 );

          swap_ranges( vec.begin(), last, vec2.begin() );

     cout << "первый контейнер после перестановки двух векторов:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

     cout << "второй контейнер после перестановки двух векторов:\n";
     copy( vec2.begin(), vec2.end(), ofile ); cout << '\n';

}
```

```
template< class InputIterator, class OutputIterator,
          class UnaryOperation >
OutputIterator
transform( InputIterator first, InputIterator last,
           OutputIterator result, UnaryOperation op );

template< class InputIterator1, class InputIterator2,
          class OutputIterator, class BinaryOperation >
OutputIterator
transform( InputIterator1 first1, InputIterator1 last,
           InputIterator2 first2, OutputIterator result,
```

Алгоритм transform()

```
           BinaryOperation bop );
```

Первый вариант `transform()` генерирует новую последовательность, применяя операцию `op` к каждому элементу из диапазона `[first,last)`. Например, если есть последовательность {0,1,1,2,3,5} и объект-функция `Double`, удваивающий свой аргумент, то в результате получим {0,2,2,4,6,10}.

Второй вариант генерирует новую последовательность, применяя бинарную операцию `bop` к паре элементов, один из которых взят из диапазона `[first1,last1)`, а второй – из последовательности, начинающейся с `first2`. Поведение программы не определено, если во второй последовательности меньше элементов, чем в первой. Например, для двух последовательностей {1,3,5,9} и {2,4,6,8} и объекта-функции `AddAndDouble`, которая складывает два элемента и удваивает их сумму, результатом будет {6,14,22,34}.

Оба варианта `transform()` помещают результирующую последовательность в контейнер с элемента, на который указывает итератор `result`. Этот итератор может адресовать и элемент любого из входных контейнеров, в таком случае исходные элементы будут заменены на результат выполнения `transform()`. Выходной итератор указывает на элемент за последним помещенным в результирующий контейнер.

```
#include <algorithm>
#include <vector>
#include <math.h>
#include <iostream.h>

/*
* печатается:
  исходный массив: 3 5 8 13 21
  преобразование элементов путем удваивания: 6 10 16 26 42
  преобразование элементов путем взятия разности: 3 5 8 13 21
*/

int double_val( int val ) { return val + val; }
int difference( int val1, int val2 ) {
    return abs( val1 - val2 ); }

int main()
{
          int ia[]  = { 3, 5, 8, 13, 21 };
          vector<int, allocator> vec( 5 );
          ostream_iterator<int> outfile( cout, " " );

          cout << "исходный массив: ";
          copy( ia, ia+5, outfile ); cout << endl;

     cout << "преобразование элементов путем удваивания: ";
          transform( ia, ia+5, vec.begin(), double_val );
          copy( vec.begin(), vec.end(), outfile ); cout << endl;

     cout << "преобразование элементов путем взятия разности: ";
          transform( ia, ia+5, vec.begin(), outfile, difference );
          cout << endl;

}
```

```
template< class ForwardIterator >
ForwardIterator
unique( ForwardIterator first,
        ForwardIterator last );
template< class ForwardIterator, class BinaryPredicate >
ForwardIterator
unique( ForwardIterator first,
```

Алгоритм unique()

```
        ForwardIterator last, BinaryPredicate pred );
```

Все группы равных соседних элементов заменяются одним. В первом варианте при сравнении используется оператор равенства, определенный для типа элементов в контейнере. Во втором варианте два элемента равны, если бинарный предикат `pred` для них возвращает `true`. Таким образом, слово `mississippi` будет преобразовано в `misisipi`. Обратите внимание, что три буквы `'i'` не являются соседними, поэтому они не заменяются одной, как и две пары несоседних `'s'`. Если нужно, чтобы все одинаковые элементы были заменены одним, придется сначала отсортировать контейнер.

На самом деле поведение unique() интуитивно не совсем очевидно и напоминает remove(). В обоих случаях размер контейнера не изменяется: каждый уникальный элемент помещается в очередную позицию, начиная с first.

В нашем примере *физически* будет получено слово misisipippi, где ppi – остаток, "отходы" алгоритма. Возвращаемый итератор указывает на начало этого остатка и обычно передается алгоритму erase() для удаления ненужных элементов. (Поскольку для встроенного массива операция erase() не поддерживается, то лучше воспользоваться алгоритмом unique_copy().)

```
template< class InputIterator, class OutputIterator >
OutputIterator
unique_copy( InputIterator first, InputIterator last,
             OutputIterator result );

template< class InputIterator, class OutputIterator,
          class BinaryPredicate >
OutputIterator
unique_copy( InputIterator first, InputIterator last,
```

Алгоритм unique_copy()

```
             OutputIterator result, BinaryPredicate pred );
```

unique_copy() копирует входной контейнер в выходной, заменяя группы одинаковых соседних элементов на один элемент с тем же значением. О том, что понимается под равными элементами, говорилось при описании алгоритма unique(). Чтобы все дубликаты были гарантированно удалены, входной контейнер необходимо предварительно отсортировать. Возвращаемый итератор указывает на элемент за последним скопированным.

```
#include <algorithm>
#include <vector>
#include <string>
#include <iterator>
#include <assert.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

void (*pfi)( int ) = print_elements;
void (*pfs)( string ) = print_elements;

int main()
{
          int ia[] = { 0, 1, 0, 2, 0, 3, 0, 4, 0, 5 };

          vector<int,allocator> vec( ia, ia+10 );
          vector<int,allocator>::iterator vec_iter;

          // последовательность не изменяется: нули не стоят рядом
     // печатается: 0 1 0 2 0 3 0 4 0 5
          vec_iter = unique( vec.begin(), vec.end() );
          for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

          // отсортировать вектор, затем применить unique:
          модифицируется
     // печатается: 0 1 2 3 4 5 2 3 4 5
          sort( vec.begin(), vec.end() );
          vec_iter = unique( vec.begin(), vec.end() );
     for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

          // удалить из контейнера ненужные элементы
     // печатается: 0 1 2 3 4 5
          vec.erase( vec_iter, vec.end() );
     for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

          string sa[] = { "enough", "is", "enough",
                    "enough", "is", "good" };

          vector<string,allocator> svec( sa, sa+6 );
          vector<string,allocator> vec_result( svec.size() );
          vector<string,allocator>::iterator svec_iter;

     sort( svec.begin(), svec.end() );
          svec_iter = unique_copy( svec.begin(), svec.end(),
                              vec_result.begin() );

     // печатается: enough good is
     for_each( vec_result.begin(), svec_iter, pfs );
          cout << "\n\n";

}
```

```
template< class ForwardIterator, class Type >
ForwardIterator
upper_bound( ForwardIterator first,
             ForwardIterator last, const Type &value );

template< class ForwardIterator, class Type, class Compare >
ForwardIterator
upper_bound( ForwardIterator first,
             ForwardIterator last, const Type &value,
```

Алгоритм upper_bound()

```
             Compare comp );
```

upper_bound() возвращает итератор, указывающий на последнюю позицию в отсортированной последовательности [first,last), в которую еще можно вставить значение value, не нарушая упорядоченности. Значения всех элементов, начиная с этой позиции и далее, будут больше, чем value. Например, если дана последовательность:

```
int ia[] = {12,15,17,19,20,22,23,26,29,35,40,51};
```

то обращение к upper_bound() с value=21 вернет итератор, указывающий на значение 22, а обращение с value=22 – на значение 23. В первом варианте для сравнения используется оператор “меньше”, определенный для типа элементов контейнера; во втором – заданная программистом операция comp.

```
#include <algorithm>
#include <vector>
#include <assert.h>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

void (*pfi)( int ) = print_elements;

int main()
{
          int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};
          vector<int,allocator> vec(ia,ia+12);

          sort(ia,ia+12);
          int *iter = upper_bound(ia,ia+12,19);
          assert( *iter == 20 );

          sort( vec.begin(), vec.end(), greater<int>() );
          vector<int,allocator>::iterator iter_vec;

          iter_vec = upper_bound( vec.begin(), vec.end(),
                              27, greater<int>() );

          assert( *iter_vec == 26 );

          // печатается: 51 40 35 29 27 26 23 22 20 19 17 15 12
          vec.insert( iter_vec, 27 );
          for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

}
```

Алгоритмы для работы с хипом

В стандартной библиотеке используется *макс-хип*. Макс-хип – это представленное в виде массива двоичное дерево, для которого значение ключа в каждом узле больше либо равно значению ключа в каждом из узлов-потомков. (Подробное обсуждение макс-хипа можно найти в [SEDGEWICK88]. Альтернативой ему является *мин-хип*, для которого значение ключа в каждом узле меньше либо равно значению ключа в каждом из узлов-потомков.) В реализации из стандартной библиотеки самое большое значение (корень дерева) всегда оказывается в начале массива. Например, приведенная последовательность букв удовлетворяет требованиям, накладываемым на хип:

```
X T O G S M N A E R A I
```

В данном примере `X` – это корневой узел, слева от него находится `T`, а справа – `O`. Обратите внимание, что потомки не обязательно должны быть упорядочены (т.е. значение в левом узле не обязано быть меньше, чем в правом). `G` и `S` – потомки узла `T`, а `M` и `N` – потомки узла `O`. Аналогично `A` и `E` – потомки `G`, `R` и `A` – потомки `S`, `I` – левый потомок `M`, а `N` – листовой узел без потомков.

Четыре обобщенных алгоритма для работы с хипом: `make_heap()`, `pop_heap()`, `push_heap()` и `sort_heap()` – поддерживают его создание и различные манипуляции. В последних трех алгоритмах предполагается, что последовательность, ограниченная

парой итераторов, – действительно хип (в противном случае поведение программы не определено). Заметим, что список нельзя использовать как контейнер для хранения хипа, поскольку он не поддерживает произвольный доступ. Встроенный массив для размещения хипа использовать можно, но в этом случае трудно применять алгоритмы pop_heap() и push_heap(), так как они требуют изменения размера контейнера. Мы опишем все четыре алгоритма, а затем проиллюстрируем их работу на примере небольшой программы.

```
template< class RandomAccessIterator >
void
make_heap( RandomAccessIterator first,
           RandomAccessIterator last );

template< class RandomAccessIterator, class Compare >
void
make_heap( RandomAccessIterator first,
           RandomAccessIterator last, Compare comp );
```

Алгоритм make_heap()

make_heap() преобразует в хип последовательность, ограниченную диапазоном [first,last). В первом варианте для сравнения используется оператор “меньше”, определенный для типа элементов контейнера, а во втором – операция comp.

```
template< class RandomAccessIterator >
void
pop_heap( RandomAccessIterator first,
          RandomAccessIterator last );

template< class RandomAccessIterator, class Compare >
void
pop_heap( RandomAccessIterator first,
          RandomAccessIterator last, Compare comp );
```

Алгоритм pop_heap()

pop_heap() в действительности не исключает наибольший элемент, а переупорядочивает хип. Он переставляет элементы в позициях first и last-1, а затем перестраивает в хип последовательность в диапазоне [first,last-1). После этого “вытолкнутый” элемент можно получить посредством функции-члена back() контейнера либо по-настоящему исключить его с помощью pop_back(). В первом варианте при сравнении используется оператор “меньше”, определенный для типа элементов контейнера, а во втором – операция comp.

```
template< class RandomAccessIterator >
void
push_heap( RandomAccessIterator first,
           RandomAccessIterator last );

template< class RandomAccessIterator, class Compare >
void
push_heap( RandomAccessIterator first,
```

Алгоритм push_heap()

```
           RandomAccessIterator last, Compare comp );
```

push_heap() предполагает, что последовательность, ограниченная диапазоном [first,last-1), – хип и что новый добавляемый к хипу элемент находится в позиции last-1. Все элементы в диапазоне [first,last) реорганизуются в новый хип. Перед вызовом push_heap() необходимо вставить новый элемент в конец контейнера, возможно, применив функцию push_back() (это показано в примере ниже). В первом варианте при сравнении используется оператор “меньше”, определенный для типа элементов контейнера; во втором – операция comp.

```
template< class RandomAccessIterator >
void
sort_heap( RandomAccessIterator first,
           RandomAccessIterator last );

template< class RandomAccessIterator, class Compare >
void
sort_heap( RandomAccessIterator first,
```

Алгоритм sort_heap()

```
           RandomAccessIterator last, Compare comp );
```

sort_heap() сортирует последовательность в диапазоне [first,last), предполагая, что это правильно построенный хип; в противном случае поведение программы не определено. (Разумеется, после сортировки хип перестает быть хипом!) В первом варианте при сравнении используется оператор “меньше”, определенный для типа элементов контейнера, а во втором – операция comp.

```
#include <algorithm>
#include <vector>
#include <assert.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

int main()
{
          int ia[] = { 29,23,20,22,17,15,26,51,19,12,35,40 };
          vector< int, allocator > vec( ia, ia+12 );


     // печатается: 51 35 40 23 29 20 26 22 19 12 17 15
          make_heap( &ia[0], &ia[12] );
     void (*pfi)( int ) = print_elements;
     for_each( ia, ia+12, pfi ); cout << "\n\n";

     // печатается: 12 17 15 19 23 20 26 51 22 29 35 40
     // минимальный хип: в корне наименьший элемент
          make_heap( vec.begin(), vec.end(), greater<int>() );
     for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

     // печатается: 12 15 17 19 20 22 23 26 29 35 40 51
          sort_heap( ia, ia+12 );
     for_each(  ia, ia+12, pfi ); cout << "\n\n";

          // добавим новый наименьший элемент
          vec.push_back( 8 );

     // печатается: 8 17 12 19 23 15 26 51 22 29 35 40 20
          // новый наименьший элемент должен оказаться в корне
          push_heap( vec.begin(), vec.end(), greater<int>() );
     for_each(  vec.begin(), vec.end(), pfi ); cout << "\n\n";

     // печатается: 12 17 15 19 23 20 26 51 22 29 35 40 8
          // наименьший элемент должен быть заменен на следующий по
          порядку

          pop_heap( vec.begin(), vec.end(), greater<int>() );
     for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

}
```

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

B

Г

Д

Е

З

И

К

Л

M

H

О

П

P

C

T

У

Ф

Х

Ц

Ч

Ш

Э

Я