

Бестселлер с 1986 года

Полностью пересмотренное и обновленное

издание под стандарт C++11!



Пятое издание

Язык программирования

C++

Базовый курс

Стенли Б. Липпман
Жози Лажойе
Барбара Э. Му

Бестселлер с 1986 года

**Полностью пересмотренное и обновленное
издание под стандарт C++11!**



Пятое издание

Язык программирования

C++

Базовый курс

Стенли Б. Липпман
Жози Лажойе
Барбара Э. Му

Язык программирования C++

Пятое издание

Посвящается Бет, благодаря которой стало возможным написание этой и всех остальных книг.

Посвящается Дэниелю и Анне, для которых возможно практически все.

Стэнли Б. Липпман

*Посвящается Марку и маме,
безграничную любовь и поддержку.*

Жози Лажойе

*Посвящается Энди, научившему меня
программированию и многому другому.*

Барбара Му

Введение

Благодаря предыдущим изданиям книги язык C++ изучило множество программистов. За истекшее время язык C++ претерпел существенные усовершенствования, а основное внимание сообщества программистов переместилось главным образом с эффективности использования аппаратных средств к эффективности программирования.

В 2011 году комитет по стандартам C++ выпустил новую основную версию стандарта ISO C++. Этот пересмотренный стандарт является последним этапом развития языка C++, его основное внимание уделено эффективности программирования. Основные задачи нового стандарта таковы.

- Сделать язык более единообразным, упростить его преподавание и изучение.
- Упростить, обезопасить и повысить эффективность использования стандартных библиотек.
- Облегчить написание эффективных абстракций и библиотек.

Это издание книги полностью пересмотрено так, чтобы использовать последний стандарт языка. Просмотрев раздел "Новые средства C++11" после оглавления, вы можете получить представление о том, насколько сильно новый стандарт повлиял на язык C++. Там перечислены только те разделы, в которых рассматривается новый материал.

Некоторые из нововведений в новом стандарте, такие как ключевое слово `auto` для выведения типов, весьма распространены. Эти средства существенно облегчают чтение кода в данном издании и делают его понятней. Программисты, конечно, могут игнорировать те средства, которые облегчают концентрацию на том, что программа призвана делать. Другие новшества, такие как интеллектуальные указатели и контейнеры с поддержкой перемещения, позволяют писать более сложные классы без необходимости справляться со сложностями управления ресурсами. В результате мы можем начать изучение создания собственных классов намного раньше, чем в предыдущем издании. Мы (и вы) больше не должны волноваться о большинстве деталей, которые стояли на нашем пути в предыдущем стандарте.



Этой пиктограммой отмечены места, в которых рассматриваются средства, определенные новым стандартом. Надеемся, что читатели, которые уже знакомы с ядром языка C++, найдут эти отметки полезными при решении, на чем сосредоточить внимание. Мы также ожидаем, что эти пиктограммы помогут объяснить сообщения об ошибках тех компиляторов, которые

могут еще не поддерживать все новые средства. Хотя практически все примеры этой книги были откомпилированы на текущем выпуске компилятора GNU, мы понимаем, что у некоторых читателей еще не будет новейшего компилятора. Даже при том, что по последнему стандарту было добавлено множество возможностей, базовый язык остается неизменным и формирует основной объем материала, который мы рассматриваем.

Для кого написана эта книга

Можно считать, что современный язык C++ состоит из трех частей.

- Низкоуровневый язык, большая часть которого унаследована от языка C.
 - Дополнительные возможности языка, позволяющие определять собственные типы данных, организовать крупномасштабные программы и системы.
 - Стандартная библиотека, использующая эти дополнительные возможности для обеспечения набора структур данных и алгоритмов.

В большинстве книг язык C++ представлен в порядке его развития. Сначала они знакомят с частью C в языке C++, а в конце книги представляются более абстрактные средства C++ как дополнительные возможности. У этого подхода есть две проблемы: читатели могут увязнуть в подробностях, унаследованных от низкоуровневого программирования, и сдаться. Те же, кто будет упорствовать в изучении, наживут плохие привычки, от которых впоследствии придется избавляться.

Мы придерживаемся противоположного подхода: с самого начала используем средства, которые позволяют программистам игнорировать детали, унаследованные от низкоуровневого программирования. Например, мы вводим и используем библиотечные типы `string` и `vector` наряду со встроенными цифровыми типами и массивами. Программы, которые используют эти библиотечные типы, проще писать, проще понимать, и ошибок в них много меньше.

Слишком часто библиотеки преподносят как "дополнительную" тему. Вместо того чтобы использовать библиотеки, во многих книгах используют низкоуровневые способы программирования с использованием указателей на символьные массивы и динамического управления памятью. Задавать правильно работать программы, которые используют эти низкоуровневые подходы, куда труднее, чем написать соответствующий код C++, используя библиотеку.

Повсюду в этой книге мы демонстрируем хороший стиль

программирования: мы хотим помочь вам выработать хорошие привычки сразу и избежать борьбы с плохими привычками впоследствии, когда вы получите более сложные навыки. Мы подчеркиваем особенно сложные моменты и предупреждаем о наиболее распространенных заблуждениях и проблемах.

Мы также объясняем причины, по которым были введены те или иные правила, а не просто принимаем их как данность. Мы полагаем, что понимание причин поможет читателю быстрей овладеть возможностями языка.

Хотя для изучения этой книги знание языка С необязательно, мы подразумеваем, что вы знакомы с программированием достаточно, чтобы написать, откомпилировать и запустить программу хотя бы на одном из современных языков. В частности, мы подразумеваем, что вы умеете использовать переменные, создавать и вызывать функции, а также использовать компилятор.

Изменения в пятом издании

Нововведением этого издания являются пиктограммы на полях, призванные помочь читателю. Язык C++ обширен, он предоставляет возможности для решения разнообразных специфических проблем программирования. Некоторые из этих возможностей весьма важны для больших групп разработчиков, но маловажны для малых проектов. В результате не каждому программисту следует знать каждую деталь каждого средства. Мы добавили эти пиктограммы, чтобы помочь читателю узнать, какие элементы могут быть изучены позже, а какие темы являются насущными.



Разделы, рассматривающие основные принципы языка, отмечены изображением человека, читающего книгу. Темы, затронутые в этих разделах, являются базовой частью языка. Все эти разделы следует прочитать и понять.



Мы также отметили те разделы, которые затрагивают дополнительные или специальные темы. Эти разделы можно пропустить или только просмотреть при первом чтении. Мы отметили такие разделы стопкой книг, указав, что на этом месте вы можете спокойно отложить книгу.

Вероятно, имеет смысл просмотреть такие разделы и узнать, какие возможности существуют. Тем не менее нет никакой причины тратить время на изучение этих тем, пока вам фактически не придется использовать в своих программах описанное средство.



Особенно сложные концепции выделены пиктограммой с изображением лупы. Надеемся, что читатели уделят время, чтобы хорошо усвоить материал, представленный в таких разделах.

Еще одна помочь читателю этой книги — обширное употребление перекрестных ссылок. Мы надеемся, что эти ссылки облегчат читателю переход в середину книги и возвращение назад к прежнему материалу, на который ссылаются более поздние примеры.

Но что остается неизменным в книге, так это четкое и ясное, корректное и полное руководство по языку C++. Мы излагаем язык, представляя наборы все более и более сложных примеров, которые объясняют его средства и демонстрируют способы наилучшего использования C++.

Структура книги

Мы начинаем с рассмотрения основ языка и библиотеки в частях I и II. Эти части содержат достаточно материала, чтобы позволить читателю писать работоспособные программы. Большинство программистов C++ должны знать все, что описано в этих частях.

Кроме обучения основам языка C++, материал частей I и II служит и другой важной цели: при использовании абстрактных средств, определенных библиотекой, вы научитесь использовать методики высокоуровневого программирования. Библиотечные средства сами являются абстрактными типами данных, которые обычно пишут на языке C++. Библиотека может быть создана с использованием тех же средств построения класса, которые доступны для любого программиста C++. Наш опыт в обучении языку C++ свидетельствует о том, что, если читатели с самого начала используют хорошо разработанные абстрактные типы, то впоследствии им проще понять, как создавать собственные типы.

Только после полного освоения основ использования библиотеки (и написания разных абстрактных программ при помощи библиотеки) мы переходим к тем средствам языка C++, которые позволяют писать

собственные абстракции. В частях III и IV главное внимание уделяется написанию абстракции в форме классов. В части III рассматриваются общие принципы, а в части IV — специализированные средства.

В части III мы рассматриваем проблемы управления копированием, а также другие способы создания классов, которые так же удобны, как и встроенные типы. Классы — это основа объектно-ориентированного и обобщенного программирования, которое также будет рассмотрено в части III. Книга заканчивается частью IV, рассматривающей средства, обычно используемые в больших и сложных системах. В приложении А приведено краткое описание библиотечных алгоритмов.

Соглашения, принятые в книге

Каждая глава завершается резюме и словарем терминов. Читатели могут использовать эти разделы как контрольный список: если вы не понимаете термин, следует повторно изучить соответствующую часть главы.

Здесь используются соглашения, общепринятые в компьютерной литературе.

- Новые термины в тексте выделяются *курсивом*. Чтобы обратить внимание читателя на отдельные фрагменты текста, также применяется курсив.
- Текст программ, функций, переменных, URL веб-страниц и другой код представлен **моноширинным шрифтом**.
- Все, что придется вводить с клавиатуры, выделено **полужирным моноширинным шрифтом**.
- Знакоместо в описаниях синтаксиса выделено курсивом. Это указывает на необходимость заменить знакоместо фактическим именем переменной, параметром или другим элементом, который должен находиться на этом месте. Например: `BINDSIZE=(максимальная ширина колонки)*(номер колонки)`.
- Пункты меню и названия диалоговых окон представлены следующим образом: **Menu Option** (Пункт меню).

Примечание о компиляторах

На момент написания этой книги (июль 2012 года) поставщики компиляторов интенсивно работали, модифицируя свои компиляторы в соответствии с последним стандартом ISO. Чаще всего мы использовали компилятор GNU версии 4.7.0. В этой книге использовано лишь несколько средств, которые в этом компиляторе еще не реализованы: наследование

конструкторов, квалификиаторы ссылок для функций-членов и библиотека регулярных выражений.

Благодарности

Мы очень благодарны за помощь в подготовке этого издания нынешним и прежним членам комитета по стандартизации: Дейв Абрахамс (Dave Abrahams), Энди Кёниг (Andy Koenig), Стефан Т. Лававей (Stephan T. Lavavej), Джейсон Меррилл (Jason Merrill), Джон Спайсер (John Spicer) и Герб Саттер (Herb Sutter). Они оказали нам неоценимую помощь в понимании некоторых нюансов нового стандарта. Мы также хотели бы поблагодарить многих других людей, которые работали над модификацией компилятора GNU и сделали стандарт реальностью.

Как и в предыдущих изданиях этой книги, мы хотели бы выразить отдельную благодарность Бъярне Страуструпу (Bjarne Stroustrup) за его неустанную работу над языком C++ и многолетнюю дружбу с авторами. Хотелось бы также поблагодарить Алекса Степанова (Alex Stepanov) за его объяснения по теме контейнеров и алгоритмов, составляющих ядро стандартной библиотеки. И наконец, сердечная благодарность членам комитета по стандарту C++ за их упорную многолетнюю работу по утверждению и усовершенствованию стандарта языка C++.

Авторы также выражают глубокую благодарность рецензентам, чьи комментарии, замечания и полезные советы помогли улучшить книгу. Спасибо Маршаллу Клоу (Marshall Clow), Джону Калбу (Jon Kalb), Невину Либеру (Nevin Liber), др. К. Л. Тондо (Dr. C. L. Tondo), Дэвиду Вандевурду (Daveed Vandevoorde) и Стиву Виноски (Steve Vinoski).

Эта книга была набрана при помощи системы LaTeX и прилагаемых к ней пакетов. Авторы выражают глубокую благодарность членам сообщества LaTeX, сделавшим доступным такой мощный инструмент.

И наконец, благодарим сотрудников издательства Addison-Wesley, которые курировали процесс публикации этой книги: Питер Гордон (Peter Gordon) — наш редактор, который предложил пересмотреть эту книгу еще раз; Ким Бодихаймер (Kim Boedigheimer) контролировал график выполнения работ; Барбара Вуд (Barbara Wood) нашла множество наших ошибок на этапе редактирования, а Элизабет Райан (Elizabeth Ryan) снова помогала авторам на протяжении всего проекта.

От издательства

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было

сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать авторам.

Мы ждем ваших комментариев. Вы можете прислать письмо по электронной почте или просто посетить наш веб-сервер, оставив на нем свои замечания. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более подходящими для вас.

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

E-mail: info@williamspublishing.com

WWW: <http://www.williamspublishing.com>

Наши почтовые адреса:

в России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1

в Украине: 03150, Киев, а/я 152

Глава 1

Первые шаги

Эта глава знакомит с большинством фундаментальных элементов языка C++: типами, переменными, выражениями, операторами и функциями. Кроме того, здесь кратко описано, как откомпилировать программу и запустить ее на выполнение.

Изучив эту главу и выполнив соответствующие упражнения, читатель будет способен написать, откомпилировать и запустить на выполнение простую программу. Последующие главы подразумевают, что вы в состоянии использовать описанные в данной главе средства и рассматривают их более подробно.

Лучше всего изучать новый язык программирования в процессе написания программ. В этой главе мы напишем простую программу для книжного магазина.

Книжный магазин хранит файл транзакций, каждая из записей которого соответствует продаже одного или нескольких экземпляров определенной книги. Каждая транзакция содержит три элемента данных:

0-201-70353-X 4 24.99

Первый элемент — это ISBN (International Standard Book Number — международный стандартный номер книги), второй — количество проданных экземпляров, последний — цена, по которой был продан каждый из этих экземпляров. Владелец книжного магазина время от времени просматривает этот файл и вычисляет для каждой книги количество проданных экземпляров, общий доход от этой книги и ее среднюю цену.

Чтобы написать эту программу, необходимо рассмотреть несколько элементарных средств языка C++. Кроме того, следует знать, как откомпилировать и запустить программу.

Хотя мы еще не разработали свою программу, несложно предположить, что для этого необходимо следующее.

- Определить переменные.
- Обеспечить ввод и вывод.
- Применить структуру для содержания данных.
- Проверить, нет ли двух записей с одинаковым ISBN.
- Использовать цикл для обработки каждой записи в файле транзакций.

Сначала рассмотрим, как эти задачи решаются средствами языка C++, а затем напишем нашу программу для книжного магазина.

1.1. Создание простой программы на языке C++

Каждая программа C++ содержит одну или несколько *функций* (function), причем одна из них обязательно имеет имя `main()`. Запуская программу C++, операционная система вызывает именно функцию `main()`. Вот простая версия функции `main()`, которая не делает ничего, кроме возвращения значения 0 операционной системе:

```
int main() {  
    return 0;  
}
```

Определение функции содержит четыре элемента: *тип возвращаемого значения* (return type), *имя функции* (function name), *список параметров* (parameter list), который может быть пустым, и *тело функции* (function body). Хотя функция `main()` является в некоторой степени особенной, мы определяем ее таким же способом, как и любую другую функцию.

В этом примере список параметров функции `main()` пуст (он представлен скобками (), в которых ничего нет). Более подробная информация о параметрах функции `main()` приведена в разделе 6.2.5.

Функция `main()` обязана иметь тип возвращаемого значения `int`, который является типом целых чисел. Тип `int` — это *встроенный тип* (built-in type) данных, такие типы определены в самом языке.

Заключительная часть определения функции, ее тело, представляет собой блок *операторов* (block of statements), который начинается открывающей *фигурной скобкой* (curly brace) и завершается закрывающей фигурной скобкой.

```
{  
    return 0;  
}
```

Единственным оператором в этом блоке является оператор `return`, который завершает код функции. Оператор `return` может также передать значение назад вызывающей стороне функции, как в данном случае. Когда оператор `return` получает значение, его тип должен быть совместим с типом возвращаемого значения функции. В данном случае типом

возвращаемого значения функции `main()` является `int`, и возвращаемое значение 0 имеет тип `int`.



Обратите внимание на точку с запятой в конце оператора `return`. Точкой с запятой отмечают конец большинства операторов языка C++. Ее очень просто пропустить, и это приводит к выдаче компилятором непонятного сообщения об ошибке.

В большинстве операционных систем возвращаемое функцией `main()` значение используется как индикатор состояния. Возвращение значения 0 свидетельствует об успехе. Любое другое значение, как правило, означает отказ, а само значение указывает на его причину.

Ключевая концепция. Типы

Типы — это одна из наиболее фундаментальных концепций в программировании. К ней мы будем возвращаться в этой книге не раз. Тип определяет и содержимое элемента данных, и операции, которые возможны с ним.

Данные, которыми манипулируют наши программы, хранятся в переменных, и у каждой переменной есть тип. Когда типом переменной по имени `v` является `T`, мы зачастую говорим, что "переменная `v` имеет тип `T`" или "`v` есть `T`".

1.1.1. Компиляция и запуск программы

Написанную программу необходимо откомпилировать. Способ компиляции программы зависит от используемой операционной системы и компилятора. Более подробную информацию о работе используемого вами компилятора можно получить в его документации или у хорошо осведомленного коллеги.

Большинство РС-ориентированных компиляторов обладают *интегрированной средой разработки* (Integrated Development Environment — IDE), которая объединяет компилятор с соответствующими средствами редактирования и отладки кода. Эти средства весьма удобны при разработке сложных программ, однако ими следует научиться пользоваться. Описание подобных систем выходит за рамки этой книги.

Большинство компиляторов, включая укомплектованные IDE, обладают интерфейсом командной строки. Если читатель не очень хорошо знаком с IDE используемого компилятора, то, возможно, имеет смысл начать с применения более простого интерфейса командной строки. Это позволит избежать необходимости сначала изучать IDE, а затем сам язык. Кроме того, хорошо понимая язык, вам, вероятно, будет проще изучить интегрированную среду разработки.

Соглашение об именовании файлов исходного кода

Используется ли интерфейс командной строки или IDE, большинство компиляторов ожидает, что исходный код программы будет храниться в одном или нескольких файлах. Файлы программ обычно называют *файлами исходного кода* (source file). На большинстве систем имя файла исходного кода заканчивается суффиксом (расширением), где после точки следует один или несколько символов. Суффикс указывает операционной системе, что файл содержит исходный код программы C++. Различные компиляторы используют разные суффиксы; к наиболее распространенным относятся .cc, .cxx, .cpp, .ср и .C.

Запуск компилятора из командной строки

При использовании интерфейса командной строки процесс компиляции, как правило, отображается в окне консоли (например, в окне оболочки (на UNIX) или в окне командной строки (на Windows)). Подразумевая, что исходный код функции `main()` находится в файле `prog1.cc`, его можно откомпилировать при помощи команды

```
$ CC prog1.cc
```

где CC — имя компилятора; \$ — системное приглашение к вводу. Компилятор создаст исполняемый файл. На операционной системе Windows этот исполняемый файл будет называться `prog1.exe`, а компиляторы UNIX имеют тенденцию помещать исполняемые программы в файлы по имени `a.out`.

Для запуска исполняемого файла под Windows достаточно ввести в командной строке имя исполняемого файла, а расширение `.exe` можно пропустить:

```
$ prog1
```

На некоторых операционных системах местоположение файла следует указать явно, даже если файл находится в текущем каталоге или папке. В таком случае применяется следующая форма записи:

```
$ .\prog1
```

Символ . , следующий за наклонной чертой, означает, что файл находится в текущем каталоге.

Чтобы запустить исполняемый файл на UNIX, мы используем полное имя файла, включая его расширение:

```
$ a.out
```

Если бы необходимо было указать расположение файла, мы использовали бы точку (.) с последующей косой чертой, означающие, что наш исполняемый файл находится в текущем каталоге:

```
$ ./a.out
```

Способ доступа к значению, возвращаемому из функции `main()`, зависит от используемой операционной системы. В обеих операционных системах (UNIX и Windows) после выполнения программы можно ввести команду `echo` с соответствующим параметром.

На UNIX для выяснения состояния выполненной программы применяется следующая команда:

```
$ echo $?
```

В операционной системе Windows для этого применяется команда

```
$ echo %ERRORLEVEL%
```

Вызов компилятора GNU или Microsoft

Конкретная команда, используемая для вызова компилятора C++, зависит от применяемой операционной системы и версии компилятора. Наибольшее распространение получили компилятор GNU и компилятор C++ из комплекта Microsoft Visual Studio. По умолчанию для вызова компилятора GNU используется команда `g++`:

```
$ g++ -o prog1 prog1.cc
```

где \$ — это системное приглашение к вводу; -o `prog1` — аргумент компилятора и имя получаемого исполняемого файла. Данная команда создает исполняемый файл по имени `prog1` или `prog1.exe`, в зависимости от операционной системы. На операционной системе UNIX исполняемые файлы не имеют расширения, а в операционной системе Windows они имеют расширение `.exe`. Если пропустить аргумент `-o prog1`, то компилятор создаст исполняемый файл по имени `a.out` (на системе UNIX) или `a.exe` (на Windows). (Примечание: в зависимости от используемого выпуска компилятора GNU, возможно, понадобится добавить аргумент `-std=c++0x`, чтобы включить поддержку C++ 11.)

Для вызова компилятора Microsoft Visual Studio 2010 используется команда `c1`:

C: \Users\me\Programs> **cl /EHsc prog1.cpp**

где C: \Users\me\Programs> — это системное приглашение к вводу; \Users\me\Programs — имя текущего каталога (или папки). Команда cl запускает компилятор, а параметр компилятора / EHsc включает стандартную обработку исключений. Компилятор Microsoft автоматически создает исполняемый файл с именем, которое соответствует первому имени файла исходного кода. У исполняемого файла будет суффикс .exe и то же имя, что и у файла исходного кода. В данном случае исполняемый файл получит имя prog1.exe.

Как правило, компиляторы способны предупреждать о проблемных конструкциях. Обычно эти возможности имеет смысл задействовать. Поэтому с компилятором GNU желательно использовать параметр -Wall, а с компиляторами Microsoft — параметр /W4.

Более подробная информация по этой теме содержится в руководстве программиста, прилагаемом к компилятору.

Упражнения раздела 1.1.1

Упражнение 1.1. Просмотрите документацию по используемому компилятору и выясните, какое соглашение об именовании файлов он использует. Откомпилируйте и запустите на выполнение программу, функция main() которой приведена в разд. 1.1.

Упражнение 1.2. Измените код программы так, чтобы функция main() возвращала значение -1. Возвращение значения -1 зачастую свидетельствует о сбое при выполнении программы. Перекомпилируйте и повторно запустите программу, чтобы увидеть, как используемая операционная система реагирует на свидетельство об отказе функции main().

1.2. Первый взгляд на ввод-вывод

В самом языке C++ никаких операторов для *ввода и вывода* (Input/Output — IO) нет. Их предоставляет *стандартная библиотека* (standard library) наряду с обширным набором подобных средств. Однако для большинства задач, включая примеры этой книги, вполне достаточно изучить лишь несколько фундаментальных концепций и простых операций.

В большинстве примеров этой книги использована *библиотека iostream*. Ее основу составляют два типа, *istream* и *ostream*, которые представляют потоки ввода и вывода соответственно. *Поток* (stream) — это последовательность символов, записываемая или читаемая из устройства ввода-вывода некоторым способом. Термин "поток" подразумевает, что символы поступают и передаются последовательно на протяжении определенного времени.

Стандартные объекты ввода и вывода

В библиотеке определены четыре объекта ввода-вывода. Для осуществления ввода используется объект *cin* (произносится "си-ин") типа *istream*. Этот объект упоминают также как *стандартный ввод* (standard input). Для вывода используется объект *cout* (произносится "си-аут") типа *ostream*. Его зачастую упоминают как *стандартный вывод* (standard output). В библиотеке определены еще два объекта типа *ostream* — это *cerr* и *clog* (произносится "си-эрр" и "си-лог" соответственно). Объект *cerr*, называемый также *стандартной ошибкой* (standard error), как правило, используется в программах для создания предупреждений и сообщений об ошибках, а объект *clog* — для создания информационных сообщений.

Как правило, операционная система ассоциирует каждый из этих объектов с окном, в котором выполняется программа. Так, при получении данных объектом *cin* оничитываются из того окна, в котором выполняется программа. Аналогично при выводе данных объектами *cout*, *cerr* или *clog* они отображаются в том же окне.

Программа, использующая библиотеку ввода-вывода

Приложению для книжного магазина потребуется объединить несколько записей, чтобы вычислить общую сумму. Сначала рассмотрим более простую, но схожую задачу — сложение двух чисел. Используя

библиотеку ввода-вывода, можно модифицировать прежнюю программу так, чтобы она запрашивала у пользователя два числа, а затем вычисляла и выводила их сумму.

```
#include <iostream>
int main() {
    std::cout << "Enter two numbers: " << std::endl;
    int v1 = 0, v2 = 0;
    std::cin >> v1 >> v2;
    std::cout << "The sum of " << v1 << " and " << v2
        << " is " << v1 + v2 << std::endl;
    return 0;
}
```

Вначале программа отображает на экране приглашение пользователю ввести два числа.

Enter two numbers:

Затем она ожидает ввода. Предположим, пользователь ввел следующие два числа и нажал клавишу <Enter>:

3 7

В результате программа отобразит следующее сообщение:

The sum of 3 and 7 is 10

Первая строка кода (`#include <iostream>`) — это *директива препроцессора* (preprocessor directive), которая указывает компилятору^[1] на необходимость включить в программу библиотеку `ostream`. Имя в угловых скобок — это *заголовок* (header). Каждая программа, которая использует средства, хранимые в библиотеке, должна подключить соответствующий заголовок. Директива `#include` должна быть написана в одной строке. То есть и заголовок, и слово `#include` должны находиться в той же строке кода. Директива `#include` должна располагаться вне тела функции. Как правило, все директивы `#include` программы располагают в начале файла исходного кода.

Запись в поток

Первый оператор в теле функции `main()` выполняет *выражение* (expression). В языке C++ выражение состоит из одного или нескольких *операндов* (operand) и, как правило, *оператора* (operator). Чтобы отобразить подсказку на стандартном устройстве вывода, в этом выражении используется *оператор вывода* (output operator), или оператор `<<`.

```
std::cout << "Enter two numbers:" << std::endl;
```

Оператор `<<` получает два операнда: левый операнд должен быть объектом класса `ostream`, а правый операнд — это подлежащее отображению значение. Оператор заносит переданное значение в объект `cout` класса `ostream`. Таким образом, результатом является объект класса `ostream`, в который записано предоставленное значение.

Выражение вывода использует оператор `<<` дважды. Поскольку оператор возвращает свой левый операнд, результат первого оператора становится левым операндом второго. В результате мы можем сцепить вместе запросы на вывод. Таким образом, наше выражение эквивалентно следующему:

```
(std::cout << "Enter two numbers:") << std::endl;
```

У каждого оператора в цепи левый операнд будет тем же объектом, в данном случае `std::cout`. Альтернативно мы могли бы получить тот же вывод, используя два оператора:

```
std::cout << "Enter two numbers:";  
std::cout << std::endl;
```

Первый оператор выводит сообщение для пользователя. Это сообщение, *строковый литерал* (*string literal*), является последовательностью символов, заключенных в парные кавычки. Текст в кавычках выводится на стандартное устройство вывода.

Второй оператор выводит `endl` — специальное значение, называемое *манипулятором* (*manipulator*). При его записи в поток вывода происходит переход на новую строку и сброс *буфера* (*buffer*), связанного с данным устройством. Сброс буфера гарантирует, что весь вывод, который программа сформировала на данный момент, будет немедленно записан в поток вывода, а не будет ожидать записи, находясь в памяти.



Во время отладки программисты зачастую добавляют операторы вывода промежуточных значений. Для таких операторов *всегда* следует применять сброс потока. Если этого не сделать, оставшиеся в буфере вывода данные в случае сбоя программы могут ввести в заблуждение разработчика, неправильно засвидетельствовав место возникновения проблемы.

Использование имен из стандартной библиотеки

Внимательный читатель, вероятно, обратил внимание на то, что в этой

программе использована форма записи `std::cout` и `std::endl`, а не просто `cout` и `endl`. Префикс `std::` означает, что имена `cout` и `endl` определены в *пространстве имен* (namespace) по имени `std`. Пространства имен позволяют избежать вероятных конфликтов, причиной которых является совпадение имен, определенных в разных библиотеках. Все имена, определенные в стандартной библиотеке, находятся в пространстве имен `std`.

Побочным эффектом применения пространств имен библиотек является то, что названия используемых пространств приходится указывать явно, например `std`. В записи `std::cout` применяется оператор *области видимости* `::` (scope operator), позволяющий указать, что здесь используется имя `cout`, которое определено в пространстве имен `std`. Как будет продемонстрировано в разделе 3.1, существует способ, позволяющий программисту избежать частого использования подробного синтаксиса.

Чтение из потока

Отобразив приглашение к вводу, необходимо организовать чтение введенных пользователем данных. Сначала следует определить *две переменные* (variable), в данном случае `v1` и `v2`, которые и будут содержать введенные данные:

```
int v1 = 0, v2 = 0;
```

Эти переменные определены как относящиеся к типу `int`, который является встроенным типом данных для целочисленных значений. Мы также *инициализируем* (initialize) их значением 0. При инициализации переменной ей присваивается указанное значение в момент создания.

Следующий оператор читает введенные пользователем данные:

```
std::cin >> v1 >> v2;
```

Оператор ввода (input operator) (т.е. оператор `>>`) ведет себя аналогично оператору вывода. Его левым операндом является объект типа `istream`, а правым операндом — объект, заполняемый данными. Он читает значение из потока, представляемого объектом типа `istream`, и сохраняет его в объекте, заданном правым операндом. Подобно оператору вывода, оператор ввода возвращает в качестве результата свой левый операнд. Другими словами, эта операция эквивалентна следующей:

```
(std::cin >> v1) >> v2;
```

Поскольку оператор возвращает свой левый operand, мы можем объединить в одном операторе последовательность из нескольких запросов на ввод данных. Наше выражение ввода читает из объекта `std::cin` два

значения, сохраняя первое в переменной `v1`, а второе в переменной `v2`. Другими словами, рассматриваемое выражение ввода выполняется как два следующих:

```
std::cin >> v1;  
std::cin >> v2;
```

Завершение программы

Теперь осталось лишь вывести результат сложения на экран.

```
std::cout << "The sum of " << v1 << " and " << v2  
      << " is " << v1 + v2 << std::endl;
```

Хоть этот оператор и значительно длиннее оператора, отобразившего приглашение к вводу, принципиально он ничем не отличается. Он передает значения каждого из своих operandов в поток стандартного устройства вывода. Здесь интересен тот факт, что не все operandы имеют одинаковый тип значений. Некоторые из них являются строковыми литералами, например "The sum of ", другие значения относятся к типу `int`, например `v1` и `v2`, а третий представляет собой результат вычисления арифметического выражения `v1 + v2`. В библиотеке определены версии операторов ввода и вывода для всех этих встроенных типов данных.

Упражнения раздела 1.2

Упражнение 1.3. Напишите программу, которая выводит на стандартное устройство вывода фразу "Hello, World".

Упражнение 1.4. Наша программа использовала оператор суммы (+) для сложения двух чисел. Напишите программу, которая использует оператор умножения (*) для вычисления произведения двух чисел.

Упражнение 1.5. В нашей программе весь вывод осуществлял один большой оператор. Перепишите программу так, чтобы для вывода на экран каждого операнда использовался отдельный оператор.

Упражнение 1.6. Объясните, является ли следующий фрагмент кода допустимым:

```
std::cout << "The sum of " << v1;  
      << " and " << v2;  
      << " is " << v1 + v2 << std::endl;
```

Если программа корректна, то что она делает? Если нет, то почему и как ее исправить?

1.3. Несколько слов о комментариях

Прежде чем перейти к более сложным программам, рассмотрим комментарии языка C++. *Комментарий* (*comment*) помогает человеку, читающему исходный текст программы, понять ее смысл. Как правило, они используются для кратких заметок об используемом алгоритме, о назначении переменных или для дополнительных разъяснений сложного фрагмента кода. Поскольку компилятор игнорирует комментарии, они никак не влияют ни на размер исполняемой программы, ни на ее поведение или производительность.

Хотя компилятор игнорирует комментарии, читатели нашего кода — напротив. Программисты, как правило, доверяют комментариям, даже когда другие части системной документации считают устаревшими. Некорректный комментарий — это еще хуже, чем отсутствие комментария вообще, поскольку он может ввести читателя в заблуждение. Когда вы модифицируете свой код, убедитесь, что обновили и комментарии!

Виды комментариев в C++

В языке C++ существуют два вида комментариев: односторонние и парные. Односторонний комментарий начинается символом двойной наклонной черты (//) и завершается в конце строки. Все, что находится справа от этого символа в текущей строке, игнорируется компилятором.

Второй тип комментария, заключенного в пару символов /* и */, унаследован от языка С. Такие комментарии начинаются символом /* и завершаются символом */. Эти комментарии способны содержать все что угодно, включая новые строки, за исключением символа */. Все, что находится между символами /* и */, компилятор считает комментарием.

В парном комментарии могут располагаться любые символы, включая символ табуляции, пробел и символ новой строки. Парный комментарий может занимать несколько строк кода, но это не обязательно. Когда парный комментарий занимает несколько строк кода, имеет смысл указать визуально, что эти строки принадлежат многострочному комментарию. Применяемый в этой книге стиль предполагает использование для обозначения внутренних строк многострочного комментария символы звездочки. Таким образом, символ звездочки в начале строки свидетельствует о ее принадлежности к многострочному комментарию (но это необязательно).

В программах обычно используются обе формы комментариев. Парные комментарии, как правило, используют для многострочных объяснений^[2], а двойную наклонную черту — для замечаний в той же строке, что и код.

```

#include <iostream>
/*
 * Пример функции main():
 * Читает два числа и отображает их сумму
 */
int main()
{
    // Предлагает пользователю ввести два числа
    std::cout << "Enter two numbers:" << std::endl;
    int v1 = 0, v2 = 0;      // переменные для хранения
    ввода
    std::cin >> v1 >> v2; // чтение ввода
    std::cout << "The sum of " << v1 << " and " << v2
                    << " is " << v1 + v2 << std::endl;
    return 0;
}

```



В этой книге комментарии выделены курсивом, чтобы отличить их от обычного кода программы. Обычно выделение текста комментариев определяется возможностями используемой среды разработки.

Парный комментарий не допускает вложения

Комментарий, который начинается символом `/*`, всегда завершается следующим символом `*/`. Поэтому один парный комментарий не может находиться в другом. Сообщение о подобной ошибке, выданное компилятором, как правило, вызывает удивление. Попробуйте, например, откомпилировать следующую программу:

```

/*
 * парный комментарий /* */ не допускает вложения
 * под "не допускает вложения" следует понимать, что
остальная часть
 * текста будет рассматриваться как программный код
*/
int main()
{

```

```
    return 0;  
}
```

Упражнения раздела 1.3

Упражнение 1.7. Попробуйте откомпилировать программу, содержащую недопустимо вложенный комментарий.

Упражнение 1.8. Укажите, какой из следующих операторов вывода (если он есть) является допустимым:

```
std::cout << /*";  
std::cout << "*/";  
std::cout << /* "*/" */;  
std::cout << /* "*/" /* "/*" */;
```

Откомпилируйте программу с этими тремя операторами и проверьте правильность своего ответа. Исправьте ошибки, сообщения о которых были получены.

1.4. Средства управления

Операторы обычно выполняются последовательно: сначала выполняется первый оператор в блоке, затем второй и т.д. Конечно, при последовательном выполнении операторов много задач не решить (включая проблему книжного магазина). Для управления последовательностью выполнения все языки программирования предоставляют операторы, обеспечивающие более сложные пути выполнения.

1.4.1. Оператор `while`

Оператор while организует итерационное (циклическое) выполнение фрагмента кода, пока его условие остается истинным. Используя оператор `while`, можно написать следующую программу, суммирующую числа от 1 до 10 включительно:

```
#include <iostream>
int main() {
    int sum = 0, val = 1;
    // продолжать выполнение цикла, пока значение val
    // не превысит 10
    while (val <= 10) {
        sum += val; // присвоить sum сумму val и sum
        ++val;       // добавить 1 к val
    }
    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

Будучи откомпилированной и запущенной на выполнение, эта программа отобразит на экране следующий результат:

```
Sum of 1 to 10 inclusive is 55
```

Как и прежде, программа начинается с включения заголовка `iostream` и определения функции `main()`. В функции `main()` определены две переменные типа `int` — `sum`, которая будет содержать полученную сумму, и `val`, которая будет содержать каждое из значений от 1 до 10. Переменной `sum` присваивается исходное значение 0, а переменной `val` — исходное значение 1.

Новой частью программы является оператор `while`, имеющий следующий синтаксис.

```
while (условие)
    оператор
```

Оператор `while` циклически выполняет *оператор*, пока *условие* остается истинным. *Условие* — это выражение, результатом выполнения которого является истина или ложь. Пока *условие* истинно, оператор выполняется. После выполнения *оператора* *условие* проверяется снова. Если *условие* остается истинным, *оператор* выполняется снова. Цикл `while` продолжается, поочередно проверяя *условие* и выполняя *оператор*, пока *условие* не станет ложью.

В этой программе использован следующий оператор `while`:

```
// продолжать выполнение цикла, пока значение val
// не превысит 10
while (val <= 10) {
    sum += val; // присвоить sum сумму val и sum
    ++val;       // добавить 1 к val
}
```

Для сравнения текущего значения переменной `val` и числа 10 условие цикла использует *оператор меньше или равно* (*оператор* `<=`). Пока значение переменной `val` меньше или равно 10, условие истинно и тело цикла `while` выполняется. В данном случае телом цикла `while` является блок, содержащий два оператора.

```
{
    sum += val; // присвоить sum сумму val и sum
    ++val;       // добавить 1 к val
}
```

Блок (*block*) — это последовательность из любого количества операторов, заключенных в фигурные скобки. Блок является оператором и может использоваться везде, где допустим один оператор. Первым в блоке является *составной оператор присвоения* (*compound assignment operator*), или *оператор присвоения с суммой* (*оператор* `+=`). Этот оператор добавляет свой правый operand к левому operandу. Это эквивалентно двум операторам: суммы и присвоения.

```
sum = sum + val; // присвоить sum сумму val и sum
```

Таким образом, первый оператор в блоке добавляет значение переменной `val` к текущему значению переменной `sum` и сохраняет результат в той же переменной `sum`.

Следующее выражение использует *префиксный оператор инкремента* (prefix increment operator) (*оператор ++*), который осуществляет приращение:

```
++val; // добавить 1 к val
```

Оператор инкремента добавляет единицу к своему operandу. Запись `++val` эквивалентна выражению `val = val + 1`.

После выполнения тела цикла `while` снова проверяет условие. Если после нового увеличения значение переменной `val` все еще меньше или равно 10, тело цикла `while` выполняется снова. Проверка условия и выполнение тела цикла продолжится до тех пор, пока значение переменной `val` остается меньше или равно 10.

Как только значение переменной `val` станет больше 10, происходит выход из цикла `while` и управление переходит к оператору, следующему за ним. В данном случае это оператор, отображающий результат на экране, за которым следует оператор `return`, завершающий функцию `main()` и саму программу.

Упражнения раздела 1.4.1

Упражнение 1.9. Напишите программу, которая использует цикл `while` для суммирования чисел от 50 до 100.

Упражнение 1.10. Кроме оператора `++`, который добавляет 1 к своему operandу, существует оператор декремента (`--`), который вычитает 1. Используйте оператор декремента, чтобы написать цикл `while`, выводящий на экран числа от десяти до нуля.

Упражнение 1.11. Напишите программу, которая запрашивает у пользователя два целых числа, а затем отображает каждое число в диапазоне, определенном этими двумя числами.

1.4.2. Оператор `for`

В рассмотренном ранее цикле `while` для управления количеством итераций использовалась переменная `val`. Мы проверяли ее значение в условии, а затем в теле цикла `while` увеличивали его.

Такая схема, подразумевающая использование переменной в условии и ее инкремент в теле столь популярна, что было разработано второе средство управления — *оператор for*, существенно сокращающий подобный код. Используя оператор `for`, можно было бы переписать код программы, суммирующей числа от 1 до 10, следующим образом:

```
#include <iostream>
int main() {
    int sum = 0;
    // сложить числа от 1 до 10 включительно
    for (int val = 1; val <= 10; ++val)
        sum += val; // эквивалентно sum = sum + val
    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

Как и прежде, определяем и инициализируем переменную `sum` нулевым значением. В этой версии мы определяем переменную `val` как часть самого оператора `for`.

```
for (int val = 1; val <= 10; ++val)
    sum += val;
```

У каждого оператора `for` есть две части: заголовок и тело. Заголовок контролирует количество раз выполнения тела. Сам заголовок состоит из трех частей: *оператора инициализации, условия и выражения*. В данном случае оператор инициализации определяет, что объекту `val` типа `int` присвоено исходное значение 1:

```
int val = 1;
```

Переменная `val` существует только в цикле `for`; ее невозможно использовать после завершения цикла. Оператор инициализации выполняется только однажды перед запуском цикла `for`.

Условие сравнивает текущее значение переменной `val` со значением 10:

```
val <= 10
```

Условие проверяется при каждом цикле. Пока значение переменной `val` меньше или равно 10, выполняется тело цикла `for`.

Выражение выполняется после тела цикла `for`. В данном случае выражение использует префиксный оператор инкремента, который добавляет 1 к значению переменной `val`:

```
++val
```

После выполнения выражения оператор `for` повторно проверяет условие. Если новое значение переменной `val` все еще меньше или равно 10, то тело цикла `for` выполняется снова. После выполнения тела значение переменной `val` увеличивается снова. Цикл продолжается до нарушения условия.

В рассматриваемом цикле `for` тело осуществляет суммирование.

```
sum += val; // эквивалентно sum = sum + val
```

В итоге оператор `for` выполняется так.

1. Создается переменная `val` и инициализируется значением 1.

2. Проверяется значение переменной `val` (меньше или равно 10). Если условие истинно, выполняется тело цикла `for`, в противном случае цикл завершается и управление переходит к оператору, следующему за ним.

3. Приращение значения переменной `val`.

4. Пока условие истинно, повторяются действия, начиная с пункта 2.

Упражнения раздела 1.4.2

Упражнение 1.12. Что делает следующий цикл `for`? Каково финальное значение переменной `sum`?

```
int sum = 0;
for (int i = -100; i <= 100; ++i)
    sum += i;
```

Упражнение 1.13. Перепишите упражнения раздела 1.4.1, используя циклы `for`.

Упражнение 1.14. Сравните циклы с использованием операторов `for` и `while` в двух предыдущих упражнениях. Каковы преимущества и недостатки каждого из них в разных случаях?

Упражнение 1.15. Напишите программы, которые содержат наиболее распространенные ошибки, обсуждаемые во врезке «Ввод конца файла с клавиатуры». Ознакомьтесь с сообщениями, выдаваемыми компилятором.

1.4.3. Ввод неизвестного количества данных

В приведенных выше разделах мы писали программы, которые суммировали числа от 1 до 10. Логическое усовершенствование этой программы подразумевало бы запрос суммируемых чисел у пользователя. В таком случае мы не будем знать, сколько чисел суммировать. Поэтому продолжим читать числа, пока будет что читать.

```
#include <iostream>
int main() {
    int sum = 0, value = 0;
    // читать данные до конца файла, вычислить сумму
    // всех значений
    while (std::cin >> value)
        sum += value; // эквивалентно sum = sum + val
```

```
    std::cout << "Sum is: " << sum << std::endl;
    return 0;
}
```

Если ввести значения **3 4 5 6**, то будет получен результат **Sum is: 18.**

Первая строка функции `main()` определяет две переменные типа `int` по имени `sum` и `value`, инициализируемые значением 0. Переменная `value` применяется для хранения чисел, вводимых в условии цикла `while`.

```
while (std::cin >> value)
```

Условием продолжения цикла `while` является выражение

```
std::cin >> value
```

Это выражение читает следующее число со стандартного устройства ввода и сохраняет его в переменной `value`. Как упоминалось в разделе 1.2, оператор ввода возвращает свой левый операнд. Таким образом, в условии фактически проверяется объект `std::cin`.

Когда объект типа `istream` используется при проверке условия, результат зависит от состояния потока. Если поток допустим, т.е. не столкнулся с ошибкой и ввод следующего значения еще возможен, это условие считается истинным. Объект типа `istream` переходит в недопустимое состояние по достижении конца файла (`end-of-file`) или при вводе недопустимых данных, например строки вместо числа. Недопустимое состояние объекта типа `istream` в условии свидетельствует о том, что оно ложно.

Таким образом, пока не достигнут конец файла (или не произошла ошибка ввода), условие остается истинным и выполняется тело цикла `while`. Тело состоит из одного составного оператора присвоения, который добавляет значение переменной `value` к текущему значению переменной `sum`. Однажды нарушение условия завершает цикл `while`. По выходе из цикла выполняется следующий оператор, который выводит значение переменной `sum`, сопровождаемое манипулятором `endl`.

Ввод конца файла с клавиатуры

Разные операционные системы используют для конца файла различные значения. Для ввода символа конца файла в операционной системе Windows достаточно нажать комбинацию клавиш `<Ctrl+z>` (удерживая нажатой клавишу `<Ctrl>`, нажать клавишу `<z>`), а затем клавишу `<Enter>` или `<Return>`. На машине с операционной системой

UNIX, включая Mac OS-X, как правило, используется комбинация клавиш <Ctrl+d>.

Возвращаясь к компиляции

Одной из задач компилятора является поиск ошибок в тексте программ. Компилятор, безусловно, не может выяснить, делает ли программа то, что предполагал ее автор, но вполне способен обнаружить ошибки в форме записи. Ниже приведены примеры ошибок, которые компилятор обнаруживает чаще всего.

Синтаксические ошибки. Речь идет о грамматических ошибках языка C++. Приведенный ниже код демонстрирует наиболее распространенные синтаксические ошибки, снабженные комментариями, которые описывают их суть.

```
// ошибка: отсутствует ')' список параметров
функции main()
int main ( {
    // ошибка: после endl используется двоеточие, а
не точка с запятой
    std::cout << "Read each file." << std::endl;
    // ошибка: отсутствуют кавычки вокруг строкового
литерала
    std::cout << Update master. << std::endl;
    // ошибка: отсутствует второй оператор вывода
    std::cout << "Write new master." std::endl;
    // ошибка: отсутствует ';' после оператора return
    return 0
}
```

Ошибки несовпадения типа. Каждый элемент данных языка C++ имеет тип. Значение 10, например, является числом типа int. Слово "привет" с парными кавычками — это строковый литерал. Примером ошибки несовпадения является передача строкового литерала функции, которая ожидает целочисленным аргументом.

Ошибки объявления. Каждое имя, используемое в программе на языке C++, должно быть вначале объявлено. Использование необъявленного имени обычно приводит к сообщению об ошибке. Типичными ошибками объявления является также отсутствие указания пространства имен, например std::, при доступе к имени, определенному в библиотеке, а также орфографические ошибки в именах идентификаторов.

```
#include <iostream>
```

```

int main( ) {
    int v1 = 0, v2 = 0;
    std::cin >> v >> v2;      // ошибка:
используется "v" вместо "v1"
    // cout не определен, должно быть std::cout
    cout << v1 + v2 << std::endl;
    return 0;
}

```

Сообщение об ошибке содержит обычно номер строки и краткое описание того, что компилятор считает неправильным. Исправлять ошибки имеет смысл в том порядке, в котором поступают сообщения о них. Зачастую одна ошибка приводит к появлению других, поэтому компилятор, как правило, сообщает о большем количестве ошибок, чем имеется фактически. Целесообразно также перекомпилировать код после устранения каждой ошибки или небольшого количества вполне очевидных ошибок. Этот цикл известен под названием "*редактирование, компиляция, отладка*" (edit-compile-debug).

Упражнения раздела 1.4.3

Упражнение 1.16. Напишите собственную версию программы, которая выводит сумму набора целых чисел, прочитанных при помощи объекта `cin`.

1.4.4. Оператор `if`

Подобно большинству языков, C++ предоставляет *оператор if*, который обеспечивает выполнение операторов по условию. Оператор *if* можно использовать для написания программы подсчета количества последовательных совпадений значений во вводе:

```

#include <iostream>
int main() {
    // currVal - подсчитываемое число; новые значения
будем читать в val
    int currVal = 0, val = 0;
    // прочитать первое число и удостовериться в
наличии данных
    // для обработки
    if (std::cin >> currVal) {
        int cnt = 1; // сохранить счет для текущего

```

значения

```
    while ( std::cin >> val) { // читать остальные
числа
        if ( val == currVal)           // если значение то же
            ++cnt;                   // добавить 1 к cnt
        else {                       // в противном случае
            // вывести счет для
            // предыдущего значения
            std::cout << currVal << " occurs "
                << ent << " times" << std::endl;
            currVal = val;           // запомнить новое
значение
            cnt = 1;                 // сбросить счетчик
        }
    } // цикл while заканчивается здесь
// не забыть вывести счет для последнего значения
std::cout << currVal << " occurs "
    << cnt << " times" << std::endl;
} // первый оператор if заканчивается здесь
return 0;
}
```

Если задать этой программе следующий ввод:

42 42 42 42 42 55 55 62 100 100 100

то результат будет таким:

```
42 occurs 5 times
55 occurs 2 times
62 occurs 1 times
100 occurs 3 times
```

Большая часть кода в этой программе должна быть уже знакома по прежним программам. Сначала определяются переменные `val` и `currVal`: `currVal` будет содержать подсчитываемое число, а переменная `val` — каждое число, читаемое из ввода. Новыми являются два оператора `if`. Первый гарантирует, что ввод не пуст.

```
if ( std::cin >> currVal) {
    // ...
} // первый оператор if заканчивается здесь
```

Подобно оператору `while`, оператор `if` проверяет условие. Условие в первом операторе `if` читает значение в переменную `currVal`. Если

чтение успешно, то условие истинно и выполняется блок кода, начинающийся с открытой фигурной скобки после условия. Этот блок завершается закрывающей фигурной скобкой непосредственно перед оператором `return`.

Как только подсчитываемое стало известно, определяется переменная `cnt`, содержащая счет совпадений данного числа. Для многократного чтения чисел со стандартного устройства ввода используется цикл `while`, подобный приведенному в предыдущем разделе.

Телом цикла `while` является блок, содержащий второй оператор `if`:

```
if (val == currVal) // если значение то же
    ++cnt;           // добавить 1 к cnt
else {                // в противном случае вывести
    счет для           // предыдущего значения
    std::cout << currVal << " occurs "
        << cnt << " times" << std::endl;
    currVal = val;     // запомнить новое значение
    cnt = 1;            // сбросить счетчик
}
```

Условие в этом операторе `if` использует для проверки равенства значений переменных `val` и `currVal` *оператор равенства* (*equality operator*) (оператор `==`). Если условие истинно, выполняется оператор, следующий непосредственно за условием. Этот оператор осуществляет инкремент значения переменной `cnt`, означая очередное повторение значения переменной `currVal`.

Если условие ложно (т.е. значения переменных `val` и `currVal` не равны), выполняется оператор после ключевого слова `else`. Этот оператор также является блоком, состоящим из оператора вывода и двух присвоений. Оператор вывода отображает счет для значения, которое мы только что закончили обрабатывать. Операторы присвоения возвращают переменной `cnt` значение 1, а переменной `currVal` — значение переменной `val`, которое ныне является новым подсчитываемым числом.



ВНИМАНИЕ

В языке C++ для присвоения используется оператор `=`, а для проверки равенства — оператор `==`. В условии могут присутствовать оба оператора. Довольно распространена ошибка, когда в условии пишут `=`, а

подразумеваю ==.

Упражнения раздела 1.4.4

Упражнение 1.17. Что произойдет, если в рассматриваемой здесь программе все введенные значения будут равны? Что если никаких совпадающих значений нет?

Упражнение 1.18. Откомпилируйте и запустите на выполнение программу этого раздела, а затем вводите только равные значения. Запустите ее снова и вводите только не повторяющиеся числа. Совпадает ли ваше предположение с реальностью?

Упражнение 1.19. Пересмотрите свою программу, написанную для упражнения раздела 1.4.1, которая выводила бы диапазон чисел, обрабатывая ввод, так, чтобы первым отображалось меньшее число из двух введенных.

Ключевая концепция. Выравнивание и форматирование кода программ C++

Оформление исходного кода программ на языке C++ не имеет жестких правил, поэтому расположение фигурных скобок, отступ, выравнивание, комментарии и разрыв строк, как правило, никак не влияет на полученную в результате компиляции программу. Например, фигурная скобка, обозначающая начало тела функции `main()`, может находиться в одной строке со словом `main` (как в этой книге), в начале следующей строки или где-нибудь дальше. Единственное требование — чтобы открывающая фигурная скобка была первым печатным символом, за исключением комментария, после списка параметров функции `main()`.

Хотя исходный код вполне можно оформлять по своему усмотрению, необходимо все же позаботиться о его удобочитаемости. Можно, например, написать всю функцию `main()` в одной длинной строке. Такая форма записи вполне допустима, но читать подобный код будет крайне неудобно.

До сих пор не стихают бесконечные дебаты по поводу наилучшего способа оформления кода программ на языках C++ и C. Авторы убеждены, что единственно правильного стиля не существует, но единообразие все же важно. Большинство программистов выравнивают элементы своих программ так же, как мы в функции `main()` и телах наших циклов. Однако в коде этой книги принято размещать фигурные

скобки, которые разграничивают функции, в собственных строках, а выравнивание составных операторов ввода и вывода осуществлять так, чтобы совпадал отступ операндов. Другие соглашения будут описаны по мере усложнения программ.

Не забывайте, что существуют и другие способы оформления кода. При выборе стиля оформления учитывайте удобочитаемость кода, а выбрав стиль, придерживайтесь его неукоснительно на протяжении всей программы.

1.5. Введение в классы

Единственное средство, которое осталось изучить перед переходом к решению проблемы книжного магазина, — это определение *структурой данных* для хранения данных транзакций. Для определения собственных структур данных язык C++ предоставляет *классы* (class). Класс определяет тип данных и набор операций, связанных с этим типом. Механизм классов — это одно из важнейших средств языка C++. Фактически основное внимание при проектировании приложения на языке C++ уделяют именно определению различных *типов классов* (class type), которые ведут себя так же, как встроенные типы данных.

В этом разделе описан простой класс, который можно использовать при решении проблемы книжного магазина. Реализован этот класс будет в следующих главах, когда читатель больше узнает о типах, выражениях, операторах и функциях.

Чтобы использовать класс, необходимо знать следующее.

1. Каково его имя?
2. Где он определен?
3. Что он делает?

Предположим, что класс для решения проблемы книжного магазина имеет имя `Sales_item`, а определен он в заголовке `Sales_item.h`.

Как уже было продемонстрировано на примере использования таких библиотечных средств, как объекты ввода и вывода, в код необходимо включить соответствующий заголовок. Точно так же заголовки используются для доступа к классам, определенным для наших собственных приложений. Традиционно имена файлов заголовка совпадают с именами определенных в них классов. У написанных нами файлов заголовка, как правило, будет суффикс `.h`, но некоторые программисты используют расширение `.h`, `.hpp` или `.hxx`. У заголовков стандартной библиотеки обычно нет никакого суффикса вообще. Компиляторы, как правило, не заботятся о форме имен файлов заголовка, но интегрированные среды разработки иногда это делают.

1.5.1. Класс `Sales_item`

Класс `Sales_item` предназначен для хранения ISBN, а также для отслеживания количества проданных экземпляров, полученной суммы и средней цены проданных книг. Не будем пока рассматривать, как эти

данные сохраняются и вычисляются. Чтобы применить класс, необходимо знать, *что* он делает, а не *как*.

Каждый класс является определением типа. Имя типа совпадает с именем класса. Следовательно, класс `Sales_item` определен как тип `Sales_item`. Подобно встроенным типам данных, вполне можно создать переменную типа класса. Рассмотрим пример.

```
Sales_item item;
```

Этот код создает объект `item` типа `Sales_item`. Как правило, об этом говорят так: создан "объект типа `Sales_item`", или "объект класса `Sales_item`", или даже "экземпляр класса `Sales_item`".

Кроме создания переменных типа `Sales_item`, с его объектами можно выполнять следующие операции.

- Вызывать функцию `isbn()`, чтобы извлечь ISBN из объекта класса `Sales_item`.
- Использовать операторы ввода (`>>`) и вывода (`<<`), чтобы читать и отображать объекты класса `Sales_item`.
- Использовать оператор присвоения (`=`), чтобы присвоить один объект класса `Sales_item` другому.
- Использовать оператор суммы (`+`), чтобы сложить два объекта класса `Sales_item`. ISBN этих двух объектов должен совпадать. Результатом будет новый объект `Sales_item` с тем же ISBN, а количество проданных экземпляров и суммарный доход будут суммой соответствующих значений его operandов.
- Использовать составной оператор присвоения (`+=`), чтобы добавить один объект класса `Sales_item` к другому.

Ключевая концепция. Определение поведения класса

Читая эти программы, очень важно иметь в виду, что все действия, которые могут быть осуществлены с объектами класса `Sales_item`, определяет его автор. Таким образом, класс `Sales_item` определяет то, что происходит при создании объекта класса `Sales_item`, а также то, что происходит при его присвоении, сложении или выполнении операторов ввода и вывода.

Автор класса вообще определяет все операции, применимые к объектам типа класса. На настоящий момент с объектами класса `Sales_item` можно выполнять только те операции, которые перечислены в этом разделе.

Чтение и запись объектов класса Sales_item

Теперь, когда известны операции, которые можно осуществлять, используя объекты класса Sales_item, можно написать несколько простых программ, использующих его. Программа, приведенная ниже, читает данные со стандартного устройства ввода в объект Sales_item, а затем отображает его на стандартном устройстве вывода.

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item book;
    // прочитать ISBN, количество проданных
    // экземпляров и цену
    std::cin >> book;
    // вывести ISBN, количество проданных экземпляров,
    // общую сумму и среднюю цену
    std::cout << book << std::endl;
    return 0;
}
```

Если ввести значения **0-201-70353-X 4 24.99**, то будет получен результат **0-201-70353-X 4 99.96 24.99**.

В вводе было указано, что продано четыре экземпляра книги по 24,99 доллара каждый, а вывод свидетельствует, что всего продано четыре экземпляра, общий доход составлял 99,96 доллара, а средняя цена на книгу получилась 24,99 доллара.

Код программы начинается двумя директивами `#include`, одна из которых имеет новую форму. Заголовки стандартной библиотеки заключают в угловые скобки (`<>`), а те, которые не являются частью библиотеки, — в двойные кавычки (`" "`).

В функции `main()` определяется объект `book`, используемый для хранения данных, читаемых со стандартного устройства ввода. Следующий оператор осуществляет чтение в этот объект, а третий оператор выводит его на стандартное устройство вывода, сопровождая манипулятором `endl`.

Суммирование объектов класса Sales_item

Немного интересней пример суммирования двух объектов класса Sales_item.

```
#include <iostream>
#include "Sales_item.h"
int main() {
    Sales_item item1, item2;
    std::cin >> item1 >> item2; // прочитать две транзакции
    std::cout << item1 + item2 << std::endl; // отобразить их сумму
    return 0;
}
```

Если ввести следующие данные:

0-201-78345-x 3 20.00

0-201-78345-x 2 25.00

то вывод будет таким:

0-201-78345-x 5 110.22

Программа начинается с включения заголовков `Sales_item` и `iostream`. Затем создаются два объекта (`item1` и `item2`) класса `Sales_item`, предназначенные для хранения транзакций. В эти объекты читаются данные со стандартного устройства ввода. Выражение вывода суммирует их и отображает результат.

Обратите внимание: эта программа очень похожа на программу, приведенную в разд 1.2: она читает два элемента данных и отображает их сумму. Отличаются они лишь тем, что в первом случае суммируются два целых числа, а во втором — два объекта класса `Sales_item`. Кроме того, сама концепция "суммы" здесь различна. В случае с типом `int` получается обычная сумма — результат сложения двух числовых значений. В случае с объектами класса `Sales_item` используется концептуально новое понятие суммы — результат сложения соответствующих компонентов двух объектов класса `Sales_item`.

Использование перенаправления файлов

Неоднократный ввод этих транзакций при проверке программы может оказаться утомительным. Большинство операционных систем поддерживает перенаправление файлов, позволяющее ассоциировать именованный файл со стандартным устройством ввода и стандартным устройством вывода:

```
$ addItems <infile >outfile
```

Здесь подразумевается, что `$` — это системное приглашение к вводу,

а наша программа суммирования была откомпилирована в исполняемый файл addItems.exe (или addItems на системе UNIX). Эта команда будет читать транзакции из файла `infile` и записывать ее вывод в файл `outfile` в текущем каталоге.

Упражнения раздела 1.5.1

Упражнение 1.20. По адресу <http://www.informit.com/title/032174113> в каталоге кода первой главы содержится копия файла `Sales_item.h`. Скопируйте этот файл в свой рабочий каталог и используйте при написании программы, которая читает набор транзакций проданных книг и отображает их на стандартном устройстве вывода.

Упражнение 1.21. Напишите программу, которая читает два объекта класса `Sales_item` с одинаковыми ISBN и вычисляет их сумму.

Упражнение 1.22. Напишите программу, читающую несколько транзакций с одинаковым ISBN и отображающую сумму всех прочитанных транзакций.

1.5.2. Первый взгляд на функции-члены

Программа суммирования объектов класса `Sales_item` должна проверять наличие у этих объектов одинаковых ISBN. Сделаем это так:

```
#include <iostream>
#include "Sales_item.h"
int main() {
    Sales_item item1, item2;
    std::cin >> item1 >> item2;
    // сначала проверить, представляют ли объекты
    item1 и item2
    // одну и ту же книгу
    if (item1.isbn() == item2.isbn()) {
        std::cout << item1 + item2 << std::endl;
        return 0; // свидетельство успеха
    } else {
        std::cerr << "Data must refer to same ISBN"
              << std::endl;
        return -1; // свидетельство отказа
    }
}
```

Различие между этой программой и предыдущей версией в операторе `if` и его ветви `else`. Даже не понимая смысла условия оператора `if`, вполне можно понять, что делает эта программа. Если условие истинно, вывод будет, как прежде, и возвратится значение `0`, означающее успех. Если условие ложно, выполняется блок ветви `else`, который выводит сообщение об ошибке и возвращает значение `-1`.

Что такое функция-член?

Условие оператора `if` вызывает *функцию-член* (member function) `isbn()`.

```
item1.isbn() == item2.isbn()
```

Функция-член — это функция, определенная в составе класса. Функции-члены называют также *методами* (method) класса.

Вызов функции-члена обычно происходит от имени объекта класса. Например, первый, левый, операнд оператора равенства использует *оператор точки* (dot operator) (*оператор .*) для указания на то, что имеется в виду "член `isbn()` объекта по имени `item1`".

```
item1.isbn
```

Точечный оператор применим только к объектам типа класса. Левый операнд должен быть объектом типа класса, а правый операнд — именем члена этого класса. Результатом точечного оператора является член класса, заданный правым операндом.

Точечный оператор обычно используется для доступа к функциям-членам при их вызове. Для вызова функции используется *оператор вызова* (call operator) (*оператор ()*). Оператор обращения — это пара круглых скобок, заключающих список *аргументов* (argument), который может быть пуст. Функция-член `isbn()` не получает аргументов.

```
item1.isbn()
```

Таким образом, это вызов функции `isbn()`, являющейся членом объекта `item1` класса `Sales_item`. Эта функция возвращает ISBN, хранящийся в объекте `item1`.

Правый operand оператора равенства выполняется тем же способом: он возвращает ISBN, хранящийся в объекте `item2`. Если ISBN совпадают, условие истинно, а в противном случае оно ложно.

Упражнения раздела 1.5.2

Упражнение 1.23. Напишите программу, которая читает несколько транзакций и подсчитывает количество транзакций для каждого ISBN.

Упражнение 1.24. Проверьте предыдущую программу, введя несколько транзакций, представляющих несколько ISBN. Записи для каждого ISBN должны быть сгруппированы.

1.6. Программа для книжного магазина

Теперь все готово для решения проблемы книжного магазина: следует прочитать файл транзакций и создать отчет, где для каждой книги будет подсчитана общая выручка, средняя цена и количество проданных экземпляров. При этом подразумевается, что все транзакции для каждого ISBN вводятся группами.

Программа объединяет данные по каждому ISBN в переменной `total` (всего). Каждая прочитанная транзакция будет сохранена во второй переменной, `trans`. В противном случае значение объекта `total` выводится на экран, а затем заменяется только что считанной транзакцией.

```
#include <iostream>
#include "Sales_item.h"
int main() {
    Sales_item total; // переменная для хранения
    // данных следующей
    // транзакции
    // прочитать первую транзакцию и удостовериться в
    // наличии данных
    // для обработки
    if (std::cin >> total) {
        Sales_item trans; // переменная для хранения
        // текущей транзакции
        // читать и обработать остальные транзакции
        while (std::cin >> trans) {
            // если все еще обрабатывается та же книга
            if (total.isbn() == trans.isbn())
                total += trans; // пополнение текущей суммы
            else {
                // отобразить результаты по предыдущей книге
                std::cout << total << std::endl;
                total = trans; // теперь total относится к
                // следующей
                // книге
            }
        }
    }
}
```

```

    }
    std::cout << total << std::endl; // отобразить
последнюю запись
} else {
    // нет ввода! Предупредить пользователя
    std::cerr << "No data?!" << std::endl;
    return -1; // свидетельство отказа
}
return 0;
}

```

Это наиболее сложная программа из рассмотренных на настоящий момент, однако все ее элементы читателю уже знакомы.

Как обычно, код начинается с подключения используемых заголовков: `iostream` (из библиотеки) и `Sales_item.h` (собственного). В функции `main()` определен объект по имени `total` (для суммирования данных по текущему ISBN). Начнем с чтения первой транзакции в переменную `total` и проверки успешности чтения. Если чтение терпит неудачу, то никаких записей нет и управление переходит к наиболее удаленному оператору `else`, код которого отображает сообщение, предупреждающее пользователя об отсутствии данных.

Если запись введена успешно, управление переходит к блоку после наиболее удаленного оператора `if`. Этот блок начинается с определения объекта `trans`, предназначенного для хранения считываемых транзакций. Оператор `while` читает все остальные записи. Как и в прежних программах, условие цикла `while` читает значения со стандартного устройства ввода. В данном случае данные читаются в объект `trans` класса `Sales_item`. Пока чтение успешно, выполняется тело цикла `while`.

Тело цикла `while` представляет собой один оператор `if`, который проверяет равенство ISBN. Если они равны, используется составной оператор присвоения для суммирования объектов `trans` и `total`. Если ISBN не равны, отображается значение, хранящееся в переменной `total`, которой затем присваивается значение переменной `trans`. После выполнения кода оператора `if` управление возвращается к условию цикла `while`, читающему следующую транзакцию, и так далее, до тех пор, пока записи не исчерпаются. После выхода из цикла `while` переменная `total` содержит данные для последнего ISBN в файле. В последнем операторе блока наиболее удаленного оператора `if` отображаются данные

последнего ISBN.

Упражнения раздела 1.6

Упражнение 1.25. Используя загруженный с веб-сайта заголовок `Sales_item.h`, откомпилируйте и запустите программу для книжного магазина, представленную в этом разделе.

Резюме

Эта глава содержит достаточно информации о языке C++, чтобы позволить писать, компилировать и запускать простые программы. Здесь было описано, как определить функцию `main()`, которую вызывает операционная система при запуске программы. Также было продемонстрировано, как определить переменные, организовать ввод и вывод данных, использовать операторы `if`, `for` и `while`. Глава завершается описанием наиболее фундаментального элемента языка C++ — класса. Здесь было продемонстрировано создание и применение объектов классов, которые были созданы кем-то другим. Определение собственных классов будет описано в следующих главах.

Термины

Аргумент (argument). Значение, передаваемое функции.

Библиотечный тип (library type). Тип, определенный в стандартной библиотеке (например, `iostream`).

Блок (block). Последовательность операторов, заключенных в фигурные скобки.

Буфер (buffer). Область памяти, используемая для хранения данных. Средства ввода (или вывода) зачастую хранят вводимые и выводимые данные в буфере, работа которого никак не зависит от действий программы. Буфера вывода могут быть сброшены явно, чтобы принудительно осуществить запись на диск. По умолчанию буфер объекта `cin` сбрасывается при обращении к объекту `cout`, а буфер объекта `cout` сбрасывается на диск по завершении программы.

Встроенный тип (built-in type). Тип данных, определенный в самом языке (например, `int`).

Выражение (expression). Наименьшая единица вычислений. Выражение состоит из одного или нескольких операндов и оператора. Вычисление выражения определяет результат. Например, сложение

целочисленных значений ($i + j$) — это арифметическое выражение, результатом которого является сумма двух значений.

Директива#include. Делает код в указанном заголовке доступным в программе.

Заголовок (header). Механизм, позволяющий сделать определения классов или других имен доступными в нескольких программах. Заголовок включается в код программы при помощи директивы #include.

Заголовокiostream. Заголовок, предоставляющий библиотечные типы для потокового ввода и вывода.

Имя функции (function name). Имя, под которым функция известна и может быть вызвана.

Инициализация (initialize). Присвоение значения объекту в момент его создания.

Класс (class). Средство определения собственной структуры данных, а также связанных с ними действий. Класс — одно из фундаментальных средств языка C++. Классами являются такие библиотечные типы, как istream и ostream.

Комментарий (comment). Игнорируемый компилятором текст в исходном коде. Язык C++ поддерживает два вида комментариев: односторонние и парные. Односторонние комментарии начинаются символом // и продолжается до конца строки. Парные комментарии начинаются символом /* и включают весь текст до заключительного символа */.

Конец файла (end-of-file). Специфический для каждой операционной системы маркер, указывающий на завершение последовательности данных файла.

Манипулятор (manipulator). Объект, непосредственно манипулирующий потоком ввода или вывода (такой, как std::endl).

Метод (method). Синоним термина *функция-член*.

Неинициализированная переменная (uninitialized variable). Переменная, которая не имеет исходного значения. Переменные типа класса, для которых не определено никакого исходного значения, инициализируются согласно определению класса. Переменные встроенного типа, определенные в функции, являются неинициализированными, если они не были инициализированы явно. Использование значения неинициализированной переменной является ошибкой. Неинициализированные переменные являются распространенной причиной ошибок.

Объект cerr. Объект типа ostream, связанный с потоком

стандартного устройства отображения сообщений об ошибке, который зачастую совпадает с потоком стандартного устройства вывода. По умолчанию запись в объект `cerr` не буферизируется. Обычно используется для вывода сообщений об ошибках и других данных, не являющихся частью нормальной логики программы.

Объект`cin`. Объект типа `istream`, обычно используемый для чтения данных со стандартного устройства ввода.

Объект`clog`. Объект типа `ostream`, связанный с потоком стандартного устройства отображения сообщений об ошибке. По умолчанию запись в объект `clog` буферизируется. Обычно используется для записи информации о ходе выполнения программы в файл журнала.

Объект`cout`. Объект типа `ostream`, используемый для записи на стандартное устройство вывода. Обычно используется для вывода данных программы.

Оператор`!=`. Не равно. Проверяет неравенство левого и правого операндов.

Оператор`()`. Оператор вызова. Пара круглых скобок `()` после имени функции. Приводит к вызову функции. Передаваемые при вызове аргументы функции указывают в круглых скобках.

Оператор (statement). Часть программы, определяющая действие, предпринимаемое при выполнении программы. Выражение, завершающееся точкой с запятой, является оператором. Такие операторы, как `if`, `for` и `while`, имеют блоки, способные содержать другие операторы.

Оператор`--`. Оператор декремента. Вычитает единицу из операнда. Например, выражение `--i` эквивалентно выражению `i = i - 1`.

Оператор`.` Точечный оператор. Получает два операнда: левый операнд — объект, правый — имя члена класса этого объекта. Оператор обеспечивает доступ к члену класса именованного объекта.

Оператор`:`. Оператор области видимости. Кроме всего прочего, оператор области видимости используется для доступа к элементам по именам в пространстве имен. Например, запись `std::cout` указывает, что используемое имя `cout` определено в пространстве имен `std`.

Оператор`++`. Оператор инкремента. Добавляет к операнду единицу. Например, выражение `++i` эквивалентно выражению `i = i + 1`.

Оператор`+=`. Составной оператор присвоения. Добавляет правый операнд к левому, а результат сохраняет в левом операнде. Например, выражение `a += b` эквивалентно выражению `a = a + b`.

Оператор<. Меньше, чем. Проверяет, меньше ли левый операнд, чем правый.

Оператор<<. Оператор вывода. Записывает правый операнд в поток вывода, указанный левым операндом. Например, выражение `cout << "hi"` передаст слово "hi" на стандартное устройство вывода. Несколько операций вывода вполне можно объединить: выражение `cout << "hi" << "bye"` выведет слово "hibye".

Оператор<=. Меньше или равно. Проверяет, меньше или равен левый операнд правому.

Оператор=. Присваивает значение правого операнда левому.

Оператор==. Равно. Проверяет, равен ли левый операнд правому.

Оператор>. Больше, чем. Проверяет, больше ли левый операнд, чем правый.

Оператор>=. Больше или равно. Проверяет, больше или равен левый операнд правому.

Оператор>>. Оператор ввода. Считывает в правый операнд данные из потока ввода, определенного левым операндом. Например, выражение `cin >> i` считывает следующее значение со стандартного устройства ввода в переменную i. Несколько операций ввода вполне можно объединить: выражение `cin >> i >> j` считывает данные сначала в переменную i, а затем в переменную j.

Операторfor. Оператор цикла, обеспечивающий итерационное выполнение. Зачастую используется для повторения вычислений определенное количество раз.

Операторif. Управляющий оператор, обеспечивающий выполнение на основании значения определенного условия. Если условие истинно (значение `true`), выполняется тело оператора if. В противном случае (значение `false`) управление переходит к оператору else.

Операторwhile. Оператор цикла, обеспечивающий итерационное выполнение кода тела цикла, пока его условие остается истинным.

Переменная (variable). Именованный объект.

Присвоение (assignment). Удаляет текущее значение объекта и заменяет его новым.

Пространство имен (namespace). Механизм применения имен, определенных в библиотеках. Применение пространств имен позволяет избежать случайных конфликтов имени. Имена, определенные в стандартной библиотеке языка C++, находятся в пространстве имен std.

Пространство имен std. Пространство имен, используемое

стандартной библиотекой. Запись `std::cout` указывает, что используемое имя `cout` определено в пространстве имен `std`.

Редактирование, компиляция, отладка (edit-compile-debug). Процесс, обеспечивающий правильное выполнение программы.

Символьный строковый литерал (character string literal). Синоним термина *строковый литерал*.

Список параметров (parameter list). Часть определения функции. Список параметров определяет аргументы, применяемые при вызове функции. Список параметров может быть пуст.

Стандартная библиотека (standard library). Коллекция типов и функций, которой должен обладать каждый компилятор языка C++. Библиотека предоставляет типы для работы с потоками ввода и вывода. Под библиотекой программисты C++ подразумевают либо всю стандартную библиотеку, либо ее часть, библиотеку типов. Например, когда программисты говорят о библиотеке `iostream`, они подразумевают ту часть стандартной библиотеки, в которой определены классы ввода и вывода.

Стандартная ошибка (standard error). Поток вывода, предназначенный для передачи сообщения об ошибке. Обычно потоки стандартного вывода и стандартной ошибки ассоциируются с окном, в котором выполняется программа.

Стандартный ввод (standard input). Поток ввода, обычно ассоциируемый с окном, в котором выполняется программа.

Стандартный вывод (standard output). Поток вывода, обычно ассоциируемый с окном, в котором выполняется программа.

Строковый литерал (string literal). Последовательность символов, заключенных в двойные кавычки (например, "a string literal").

Структура данных (data structure). Логическое объединение типов данных и возможных для них операций.

Тело функции (function body). Блок операторов, определяющий выполняемые функцией действия.

Тип `istream`. Библиотечный тип, обеспечивающий потоковый ввод.

Тип `ostream`. Библиотечный тип, обеспечивающий потоковый вывод.

Тип возвращаемого значения (return type). Тип возвращенного функцией значения.

Тип класса (class type). Тип, определенный классом. Имя типа совпадает с именем класса.

Условие (condition). Выражение, результатом которого является

логическое значение `true` (истина) или `false` (ложь). Нуль соответствует значению `false`, а любой другой — значению `true`.

Файл исходного кода (source file). Термин, используемый для описания файла, который содержит текст программы на языке C++.

Фигурная скобка (curly brace). Фигурные скобки разграничивают блоки кода. Открывающая фигурная скобка `{`) начинает блок, а закрывающая `}` завершает его.

Функция (function). Именованный блок операторов.

Функция`main()`. Функция, вызываемая операционной системой при запуске программы C++. У каждой программы должна быть одна и только одна функция по имени `main()`.

Функция-член (member function). Операция, определенная классом. Как правило, функции-члены применяются для работы с определенным объектом.

Часть I

Основы

Все широко распространенные языки программирования предоставляют единый набор средств, отличающийся лишь специфическими подробностями конкретного языка. Понимание подробностей того, как язык предоставляет эти средства, является первым шагом к овладению данным языком. К наиболее фундаментальным из этих общих средств относятся приведенные ниже.

- Встроенные типы данных (например, целые числа, символы и т.д.).
- Переменные, позволяющие присваивать имена используемым объектам.
- Выражения и операторы, позволяющие манипулировать значениями этих типов.
- Управляющие структуры, такие как `if` или `while`, обеспечивающие условное и циклическое выполнение наборов действий.
- Функции, позволяющие обратиться к именованным блокам действий.

Большинство языков программирования дополняет эти основные средства двумя способами: они позволяют программистам дополнять язык, определяя собственные типы, а также использовать библиотеки, в которых определены полезные функции и типы, отсутствующие в базовом языке.

В языке C++, как и в большинстве языков программирования, допустимые для объекта операции определяет его тип. То есть оператор будет допустимым или недопустимым в зависимости от типа используемого объекта. Некоторые языки, например Smalltalk и Python, проверяют используемые в выражениях типы во время выполнения программы. В отличие от них, язык C++ осуществляет контроль типов данных статически, т.е. соответствие типов проверяется во время компиляции. Как следствие, компилятор требует сообщить ему тип каждого используемого в программе имени, прежде чем оно будет применено.

Язык C++ предоставляет набор встроенных типов данных, операторы для манипулирования ими и небольшой набор операторов для управления процессом выполнения программы. Эти элементы формируют алфавит, при помощи которого можно написать (и было написано) множество больших и сложных реальных систем. На этом базовом уровне язык C++ довольно прост. Его потрясающая мощь является результатом поддержки

механизмов, которые позволяют программисту самостоятельно определять новые структуры данных. Используя эти средства, программисты могут приспособить язык для собственных целей без участия его разработчиков и необходимости ожидать, пока они удовлетворят появившиеся потребности.

Возможно, важнейшим компонентом языка C++ является класс, который позволяет программистам определять собственные типы данных. В языке C++ такие типы иногда называют "типами класса", чтобы отличить их от базовых типов, встроенных в сам язык. Некоторые языки программирования позволяют определять типы, способные содержать только данные. Другие, подобно языку C++, позволяют определять типы, в состав которых можно включить операции, выполняемые с этими данными. Одна из главных задач проекта C++ заключалась в предоставлении программистам возможности самостоятельно определять типы данных, которые будут так же удобны, как и встроенные. Стандартная библиотека языка C++ использует эту возможность для реализации обширного набора классов и связанных с ними функций.

Первым шагом по овладению языком C++ является изучение его основ и библиотеки — такова тема части I, "Основы". В главе 2 рассматриваются встроенные типы данных, а также обсуждается механизм определения новых, собственных типов. В главе 3 описаны два фундаментальных библиотечных типа: `string` (строка) и `vector` (вектор). В этой же главе рассматриваются массивы, представляющие собой низкоуровневую структуру данных, встроенную в язык C++, и множество других языков. Главы 4-6 посвящены выражениям, операторам и функциям. Завершается часть главой 7 демонстрирующей основы построения собственных типов классов. Как мы увидим, в определении собственных типов примиряется все, что мы изучили до сих пор, поскольку написание класса подразумевает использование всех средств, частично раскрытых в части I.

Глава 2

Переменные и базовые типы

Типы данных — это основа любой программы: они указывают, что именно означают эти данные и какие операции с ними можно выполнять.

У языка C++ обширная поддержка таких типов. В нем определено несколько базовых типов: символы, целые числа, числа с плавающей запятой и т.д. Язык предоставляет также механизмы, позволяющие программисту определять собственные типы данных. В библиотеке эти механизмы использованы для определения более сложных типов, таких как символьные строки переменной длины, векторы и т.д. В этой главе рассматриваются встроенные типы данных и основы применения более сложных типов.

Тип определяет назначение данных и операции, которые с ними можно выполнять. Например, назначение простого оператора `i = i + j;` полностью зависит от типов переменных `i` и `j`. Если это целые числа, данный оператор представляет собой обычное арифметическое сложение. Но если это объекты класса `Sales_item`, то данный оператор суммирует их компоненты (см раздел 1.5.1).



2.1. Простые встроенные типы

В языке C++ определен набор базовых типов, включая *арифметические типы* (arithmetic type), и специальный тип `void`. Арифметические типы представляют символы, целые числа, логические значения и числа с плавающей запятой. С типом `void` не связано значений, и применяется он только при некоторых обстоятельствах, чаще всего как тип возвращаемого значения функций, которые не возвращают ничего.

2.1.1. Арифметические типы

Есть две разновидности арифметических типов: *целочисленные типы* (включая символьные и логические типы) и *типы с плавающей запятой*.

Размер (т.е. количество битов) арифметических типов зависит от конкретного компьютера. Стандарт гарантирует минимальные размеры, перечисленные в табл. 2.1. Однако компиляторы позволяют использовать для этих типов большие размеры. Поскольку количество битов не постоянно, значение одного типа также может занимать в памяти больше или меньше места.

Таблица 2.1. Арифметические типы языка C++

Тип	Значение	Минимальный размер
<code>bool</code>	Логический тип	Не определен
<code>char</code>	Символ	8 битов
<code>wchar_t</code>	Широкий символ	16 битов
<code>char16_t</code>	Символ Unicode	16 битов
<code>char32_t</code>	Символ Unicode	32 бита
<code>short</code>	Короткое целое число	16 битов
<code>int</code>	Целое число	16 битов
<code>long</code>	Длинное целое число	32 бита
<code>long long</code>	Длинное целое число	64 бита
<code>float</code>	Число с плавающей запятой одинарной точности	6 значащих цифр
<code>double</code>	Число с плавающей запятой двойной точности	10 значащих цифр

`long
double`

Число с плавающей запятой повышенной
точности

10 значащих цифр

Тип `bool` представляет только значения `true` (истина) и `false` (ложь).

Существует несколько символьных типов, большинство из которых предназначено для поддержки национальных наборов символов. Базовый символьный тип, `char`, гарантировано велик, чтобы содержать числовые значения, соответствующие символам базового набора символов машины. Таким образом, тип `char` имеет тот же размер, что и один байт на данной машине.

Остальные символьные типы, `wchar_t`, `char16_t` и `char32_t`, используются для расширенных наборов символов. Тип `wchar_t` будет достаточно большим, чтобы содержать любой символ в наибольшем расширенном наборе символов машины. Типы `char16_t` и `char32_t` предназначены для символов Unicode. (Unicode — это стандарт для представления символов, используемых, по существу, в любом языке.)



Остальные целочисленные типы представляют целочисленные значения разных размеров. Язык C++ гарантирует, что тип `int` будет по крайней мере не меньше типа `short`, а тип `long long` — не меньше типа `long`. Тип `long long` введен новым стандартом.

Машинный уровень представления встроенных типов

Компьютеры хранят данные как последовательность битов, каждый из которых содержит 0 или 1:

`00011011011100010110010000111011 ...`

Большинство компьютеров оперируют с памятью, разделенной на порции, размер которых в битах кратен степеням числа 2. Наименьшая порция адресуемой памяти называется *байтом* (byte). Основная единица хранения, обычно в несколько байтов, называется *словом* (word). В языке C++ байт содержит столько битов, сколько необходимо для содержания символа в базовом наборе символов машины. На большинстве компьютеров байт содержит 8 битов, а слово — 32 или 64 бита, т.е. 4 или 8 байтов.

У большинства компьютеров каждый байт памяти имеет номер, называемый *адресом* (address). На машине с 8-битовыми байтами и 32-битовыми словами слова в памяти можно было бы представить

следующим образом:

736424	0	0	1	1	1	0	1	1
736425	0	0	0	1	1	0	1	1
736426	0	1	1	1	0	0	0	1
736427	0	1	1	0	0	1	0	0

Слева представлен адрес байта, а 8 битов его значения — справа.

При помощи адреса можно обратиться к любому из байтов, а также к набору из нескольких байтов, начинающемуся с этого адреса. В этом случае говорят о доступе к байту по адресу 736424 или о байте, хранящемуся по адресу 736426. Чтобы получить представление о значении в области памяти по данному адресу, следует знать тип хранимого в ней значения. Именно тип определяет количество используемых битов и то, как эти биты интерпретировать.

Если известно, что объект в области по адресу 736424 имеет тип `float`, и если тип `float` на этой машине хранится в 32 битах, то известно и то, что объект по этому адресу охватывает все слово. Значение этого числа зависит от того, как именно машина хранит числа с плавающей запятой. Но если объект в области по адресу 736424 имеет тип `unsigned char`, то на машине, использующей набор символов ISO-Latin-1, этот байт представляет точку с запятой.

Типы с плавающей точкой представляют значения с одиночной, двойной и расширенной точностью. Стандарт определяет минимальное количество значащих цифр. Большинство компиляторов обеспечивает большую точность, чем минимально определено стандартом. Как правило, тип `float` представляется одним словом (32 бита), тип `double` — двумя словами (64 бита), а тип `long double` — тремя или четырьмя словами (96 или 128 битов). Типы `float` и `double` обычно имеют примерно по 7 и 16 значащих цифр соответственно. Тип `long double` зачастую используется для адаптации чисел с плавающей запятой аппаратных средств специального назначения; его точность, вероятно, также зависит от конкретной реализации этих средств.

Знаковые и беззнаковые типы

За исключением типа `bool` и расширенных символьных типов целочисленные типы могут быть *знаковыми* (`signed`) или *беззнаковыми* (`unsigned`). Знаковый тип способен представлять отрицательные и положительные числа (включая нуль); а беззнаковый тип — только

положительные числа и нуль.

Типы `int`, `short`, `long` и `long long` являются знаковыми. Соответствующий беззнаковый тип получают добавлением части `unsigned` к названию такого типа, например `unsigned long`. Тип `unsigned int` может быть сокращен до `unsigned`.

В отличие от других целочисленных типов, существуют три разновидности базового типа `char`: `char`, `signed char` и `unsigned char`. В частности, тип `char` отличается от типа `signed char`. На три символьных типа есть только два представления: знаковый и беззнаковый. Простой тип `char` использует одно из этих представлений. Какое именно, зависит от компилятора.

В беззнаковом типе все биты представляют значение. Например, 8-битовый тип `unsigned char` может содержать значения от 0 до 255 включительно.

Стандарт не определяет представление знаковых типов, но он указывает, что диапазон должен быть поровну разделен между положительными и отрицательными значениями. Следовательно, 8-битовый тип `signed char` гарантированно будет в состоянии содержать значения от -127 до 127; большинство современных машин использует представления, позволяющие содержать значения от -128 до 127.

Совет. Какой тип использовать

Подобно языку C, язык C++ был разработан так, чтобы по необходимости программа могла обращаться непосредственно к аппаратным средствам. Поэтому арифметические типы определены так, чтобы соответствовать особенностям различных аппаратных средств. В результате количество возможных арифметических типов в языке C++ огромно. Большинство программистов, желая избежать этих сложностей, ограничивают количество фактически используемых ими типов. Ниже приведено несколько эмпирических правил, способных помочь при выборе используемого типа.

- Используйте беззнаковый тип, когда точно знаете, что значения не могут быть отрицательными.
- Используйте тип `int` для целочисленной арифметики. Тип `short` обычно слишком мал, а тип `long` на практике зачастую имеет тот же размер, что и тип `int`. Если ваши значения больше, чем минимально гарантирует тип `int`, то используйте тип `long long`.
- Не используйте базовый тип `char` и тип `bool` в арифметических

выражениях. Используйте их только для хранения символов и логических значений. Вычисления с использованием типа `char` особенно проблематичны, поскольку на одних машинах он знаковый, а на других беззнаковый. Если необходимо маленькое целое число, явно определите тип как `signed char` или `unsigned char`.

- Используйте тип `double` для вычислений с плавающей точкой. У типа `float` обычно недостаточно точности, а различие в затратах на вычисления с двойной и одинарной точностью незначительны. Фактически на некоторых машинах операции с двойной точностью осуществляются быстрее, чем с одинарной. Точность, предоставляемая типом `long double`, обычно чрезмерна и не нужна, а зачастую влечет значительное увеличение продолжительности выполнения.

Упражнения раздела 2.1.1

Упражнение 2.1. Каковы различия между типами `int`, `long`, `long long` и `short`? Между знаковыми и беззнаковыми типами? Между типами `float` и `double`?

Упражнение 2.2. Какие типы вы использовали бы для коэффициента, основной суммы и платежей при вычислении выплат по закладной? Объясните, почему вы выбрали каждый из типов?



2.1.2. Преобразование типов

Тип объекта определяет данные, которые он может содержать, и операции, которые с ним можно выполнять. Среди операций, поддерживаемых множеством типов, есть возможность *преобразовать* (convert) объект данного типа в другой, связанный тип.

Преобразование типов происходит автоматически, когда объект одного типа используется там, где ожидается объект другого типа. Более подробная информация о преобразованиях приведена в разделе 4.11, а пока имеет смысл понять, что происходит при присвоении значения одного типа объекту другого.

Когда значение одного арифметического типа присваивается другому

```
bool b = 42;           // b содержит true
int i = b;             // i содержит значение 1
i = 3.14;              // i содержит значение 3
```

```
double pi = i;           // pi содержит значение 3.0
unsigned char c = -1;    // при 8-битовом char
содержит значение 255
signed char c2 = 256;   // при 8-битовом char
значение c2 не определено
```

происходящее зависит от диапазона значений, поддерживаемых типом.

- Когда значение одного из не логических арифметических типов присваивается объекту типа `bool`, результат будет `false`, если значением является `0`, а в противном случае — `true`.

- Когда значение типа `bool` присваивается одному из других арифметических типов, будет получено значение `1`, если логическим значением было `true`, и `0`, если это было `false`.

- Когда значение с плавающей точкой присваивается объекту целочисленного типа, оно усекается до части перед десятичной точкой.

- Когда целочисленное (интегральное) значение присваивается объекту типа с плавающей точкой, дробная часть равна нулю. Если у целого числа больше битов, чем может вместить объект с плавающей точкой, то точность может быть потеряна.

- Если объекту беззнакового типа присваивается значение не из его диапазона, результатом будет остаток от деления по модулю значения, которые способен содержать тип назначения. Например, 8-битовый тип `unsigned char` способен содержать значения от `0` до `255` включительно. Если присвоить ему значение вне этого диапазона, то компилятор присвоит ему остаток от деления по модулю `256`. Поэтому в результате присвоения значения `-1` переменной 8-битового типа `unsigned char` будет получено значение `255`.

- Если объекту знакового типа присваивается значение не из его диапазона, результат оказывается *не определен*. В результате программа может сработать нормально, а может и отказаться или задействовать неверное значение.

Совет. Избегайте неопределенного и машинно-зависимого поведения

Результатом неопределенного поведения являются такие ошибки, которые компилятор не обязан (а иногда и не в состоянии) обнаруживать. Даже если код компилируется, то программа с неопределенным выражением все равно ошибочна.

К сожалению, программы, характеризующиеся неопределенным

поведением на некоторых компиляторах и при некоторых обстоятельствах, могут работать вполне нормально, не проявляя проблему. Но нет никаких гарантий, что та же программа, откомпилированная на другом компиляторе или даже на следующей версии данного компилятора, продолжит работать правильно. Нет даже гарантий того, что, нормально работая с одним набором данных, она будет нормально работать с другим.

Аналогично в программах нельзя полагаться на машинно-зависимое поведение. Не стоит, например, надеяться на то, что переменная типа `int` имеет фиксированный, заранее известный размер. Такие программы называют *непереносимыми* (*nonportable*). При переносе такой программы на другую машину любой полагающийся на машинно-зависимое поведение код, вероятней всего, сработает неправильно, поэтому его придется искать и исправлять. Поиск подобных проблем в ранее нормально работавшей программе, мягко говоря, не самая приятная работа.

Компилятор применяет эти преобразования типов при использовании значений одного арифметического типа там, где ожидается значение другого арифметического типа. Например, при использовании значения, отличного от логического в условии, арифметическое значение преобразуется в тип `bool` таким же образом, как при присвоении арифметического значения переменной типа `bool`:

```
int i = 42;
if (i) // условие рассматривается как истинное
    i = 0;
```

При значении 0 условие будет ложным, а при всех остальных (отличных от нуля) — истинным.

К тому же при использовании значения типа `bool` в арифметическом выражении оно всегда преобразуется в 0 или 1. В результате применение логического значения в арифметическом выражении является неправильным.



Выражения, задействующие беззнаковые типы

Хотя мы сами вряд ли преднамеренно присвоим отрицательное значение объекту беззнакового типа, мы можем (причем слишком легко) написать код, который сделает это неявно. Например, если использовать

значения типа `unsigned` и `int` в арифметическом выражении, значения типа `int` обычно преобразуются в тип `unsigned`. Преобразование значения типа `int` в `unsigned` выполняется таким же способом, как и при присвоении:

```
unsigned u = 10;
int i = -42;
std::cout << i + i << std::endl; // выводит -84
std::cout << u + i << std::endl; // при 32-битовом
int,
// выводит
4294967264
```

Во втором выражении, прежде чем будет осуществлено сложение, значение `-42` типа `int` преобразуется в значение типа `unsigned`. Преобразование отрицательного числа в тип `unsigned` происходит точно так же, как и при попытке присвоить это отрицательное значение объекту типа `unsigned`. Произойдет "обращение значения" (wrap around), как было описано выше.

При вычитании значения из беззнакового объекта, независимо от того, один или оба операнда являются беззнаковыми, следует быть уверенным том, что результат не окажется отрицательным:

```
unsigned u1 = 42, u2 = 10;
std::cout << u1 - u2 << std::endl; // ok: результат
32
std::cout << u2 - u1 << std::endl; // ok: но с
обращением значения
```

Тот факт, что беззнаковый объект не может быть меньше нуля, влияет на способы написания циклов. Например, в упражнениях раздела 1.4.1 (стр. 39) следовало написать цикл, который использовал оператор декремента для вывода чисел от 10 до 0. Написанный вами цикл, вероятно, выглядел примерно так:

```
for (int i = 10; i >= 0; --i)
    std::cout << i << std::endl;
```

Казалось бы, этот цикл можно переписать, используя тип `unsigned`. В конце концов, мы не планируем выводить отрицательные числа. Однако это простое изменение типа приведет к тому, что цикл никогда не закончится:

```
// ОШИБКА: и никогда не сможет стать меньше 0;
условие
```

```
// навсегда останется истинным
for (unsigned u = 10; u >= 0; --u)
    std::cout << u << std::endl;
```

Рассмотрим, что будет, когда *u* станет равно 0. На этой итерации отображается значение 0, а затем выполняется выражение цикла `for`. Это выражение, `--u`, вычитает 1 из *u*. Результат, -1, недопустим для беззнаковой переменной. Как и любое другое значение, не попадающее в диапазон допустимых, это будет преобразовано в беззнаковое значение. При 32-разрядном типе `int` результат выражения `--u` при *u* равном 0 составит 4294967295.

Исправить этот код можно, заменив цикл `for` циклом `while`, поскольку последний осуществляет декремент прежде (а не после) отображения значения:

```
unsigned u = 11; // начать цикл с элемента на один
больше
```

// первого, подлежащего
отображению

```
while (u > 0) {
    --u; // сначала декремент, чтобы последняя
итерация отобразила 0
    std::cout << u << std::endl;
}
```

Цикл начинается с декремента значения управляющей переменной цикла. В начале последней итерации переменная *u* будет иметь значение 1, а после декремента мы отобразим значение 0. При последующей проверке условия цикла `while` значением переменной *u* будет 0, и цикл завершится. Поскольку декремент осуществляется сначала, переменную *u* следует инициализировать значением на единицу больше первого подлежащего отображению значения. Следовательно, чтобы первым отображаемым значением было 10, переменную *u* инициализируем значением 11.

Внимание! Не смешивайте знаковые и беззнаковые типы

Выражения, в которых смешаны знаковые и беззнаковые типы, могут приводить к удивительным результатам, когда знаковое значение оказывается негативным. Важно не забывать, что знаковые значения автоматически преобразовываются в беззнаковые. Например, в таком выражении, как `a * b`, если *a* содержит значение -1, а *b* значение 1 и

обе переменные имеют тип `int`, ожидается результат `-1`. Но если переменная `a` имеет тип `int`, а переменная `b` — тип `unsigned`, то значение этого выражения будет зависеть от количества битов, занимаемых типом `int` на данной машине. На нашей машине результатом этого выражения оказалось `4294967295`.

Упражнения раздела 2.1.2

Упражнение 2.3. Каков будет вывод следующего кода?

```
unsigned u = 10, u2 = 42;
std::cout << u2 - u << std::endl;
std::cout << u - u2 << std::endl;
int i = 10, i2 = 42;
std::cout << i2 - i << std::endl;
std::cout << i - i2 << std::endl;
std::cout << i - u << std::endl;
std::cout << u - i << std::endl;
```

Упражнение 2.4. Напишите программу для проверки правильности ответов. При неправильных ответах изучите этот раздел еще раз.

2.1.3. Литералы

Такое значение, как 42, в коде программы называется *литералом* (literal), поскольку его значение самоочевидно. У каждого литерала есть тип, определяемый его формой и значением.

Целочисленные литералы и литералы с плавающей запятой

Целочисленный литерал может быть в десятичной, восьмеричной или шестнадцатеричной форме. Целочисленные литералы, начинающиеся с нуля (0), интерпретируются как восьмеричные, а начинающиеся с 0x или 0X — как шестнадцатеричные. Например, значение 20 можно записать любым из трех следующих способов.

```
20    // десятичная форма  
024   // восьмеричная форма  
0x14  // шестнадцатеричная форма
```

Тип целочисленного литерала зависит от его значения и формы. По умолчанию десятичные литералы считаются знаковыми, а восьмеричные и шестнадцатеричные литералы могут быть знаковыми или беззнаковыми. Для десятичного литерала принимается наименьший тип, `int`, `long`, или `long long`, подходящий для его значения (т.е. первый подходящий в этом списке). Для восьмеричных и шестнадцатеричных литералов принимается наименьший тип, `int`, `unsigned int`, `long`, `unsigned long`, `long long` или `unsigned long long`, подходящий для значения литерала. Не следует использовать литерал, значение которого слишком велико для наибольшего соответствующего типа. Нет литералов типа `short`. Как можно заметить в табл. 2.2, значения по умолчанию можно переопределить при помощи суффикса.

Хотя целочисленные литералы могут иметь знаковый тип, с технической точки зрения значение десятичного литерала никогда не бывает отрицательным числом. Если написать нечто, выглядящее как отрицательный десятичный литерал, например `-42`, то знак "минус" *не будет* частью литерала. Знак "минус" — это оператор, который инвертирует знак своего операнда (литерала).

Литералы с плавающей запятой включают либо десятичную точку, либо экспоненту, определенную при помощи экспоненциального представления. Экспонента в экспоненциальном представлении обозначается символом `E` или `e`:

```
3.14159 3.14159E0 0. 0e0 .001
```

По умолчанию литералы с плавающей запятой имеют тип `double`. Используя представленные в табл. 2.2 суффиксы, тип умолчанию можно переопределить.

Символьные и строковые литералы

Символ, заключенный в одинарные кавычки, является литералом типа `char`. Несколько символов, заключенных в парные кавычки, являются строковым литералом:

```
'a'           // символный литерал  
"Hello World!" // строковый литерал
```

Типом строкового литерала является массив константных символов. Этот тип обсуждается в разделе 3.5.4. К каждому строковому литералу компилятор добавляет *нулевой символ* (null character) ('`\0`'). Таким образом, реальная величина строкового литерала на единицу больше его видимого размера. Например, литерал '`A`' представляет один символ `A`, тогда как строковый литерал "`A`" представляет массив из двух символов, символа `A` и нулевого символа.

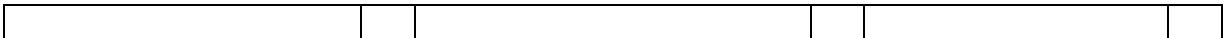
Два строковых литерала, разделенных пробелами, табуляцией или символом новой строки, конкатенируются в единый литерал. Такую форму литерала используют, если необходимо написать слишком длинный текст, который неудобно располагать в одной строке.

```
// многострочный литерал  
std::cout << "a really, really long string literal  
"  
                  "that spans two lines" << std::endl;
```

Управляющие последовательности

У некоторых символов, таких как возврат на один символ или управляющий символ, нет видимого изображения. Такие символы называют *непечатаемыми* (nonprintable character). Другие символы (одиночные и парные кавычки, вопросительный знак и наклонная черта влево) имеют в языке специальное назначение. В программах нельзя использовать ни один из этих символов непосредственно. Для их представления как символов используется *управляющая последовательность* (escape sequence), начинающаяся с символа наклонной черты влево.

В языке C++ определены следующие управляющие последовательности.



Новая строка (newline)	\n	Горизонтальная табуляция (horizontal tab)	\t	Оповещение, звонок (alert)	\a
Вертикальная табуляция (vertical tab)	\v	Возврат на один символ (backspace)	\b	Двойная кавычка (double quote)	\"
Наклонная черта влево (backslash)	\\\	Вопросительный знак (question mark)	\?	Одинарная кавычка (single quote)	\'
Возврат каретки (carriage return)	\r	Прогон страницы (formfeed)	\f		

Управляющую последовательность используют как единый символ:

```
std::cout << '\n'; // отобразить новую строку
std::cout << "\tHi! \n"; // отобразить табуляцию,
                           // текст "Hi!" и новую
```

строка

Можно также написать обобщенную управляющую последовательность, где за \x следует одна или несколько шестнадцатеричных цифр или за \x следует одна, две или три восьмеричные цифры. Так можно отобразить символ по его числовому значению. Вот несколько примеров (подразумевается использование набора символов Latin-1):

```
\7 (оповещение) \12 (новая строка) \40 (пробел)
\0 (нулевой символ) \115 (символ 'M') \x4d (символ 'M')
```

Как и управляющие последовательности, определенные языком, такой синтаксис можно использовать вместо любого другого символа:

```
std::cout << "Hi \x4dO\115! \n"; // выводит Hi MOM!
и новую строку
std::cout << '\115' << '\n'; // выводит M и
новую строку
```

Обратите внимание: если символ \x сопровождается более чем тремя восьмеричными цифрами, то ассоциируются с ним только первые три. Например, литерал "\1234" представляет два символа: символ, представленный восьмеричным значением 123, и символ 4. Форма \x, напротив, использует все последующие шестнадцатеричные цифры; литерал "\x1234" представляет один 16-разрядный символ, состоящий из битов, соответствующих этим четырем шестнадцатеричным цифрам. Поскольку большинство машин использует 8-битовые символы, подобные значения вряд ли будут полезны. Обычно шестнадцатеричные символы с более чем 8 битами используются для расширенных наборов символов с применением одного из префиксов, приведенных в табл. 2.2.

Определение типа литерала

При помощи суффикса или префикса, представленного в табл. 2.2, можно переопределить заданный по умолчанию тип целого числа, числа с плавающей запятой или символьного литерала.

```
L' a'      // литерал типа wchar_t (широкий символ)
u8"hi!"    // строковый литерал utf-8 (8-битовая
кодировка Unicode)
42ULL      // целочисленный беззнаковый литерал, тип
unsigned long long
1E-3F       // литерал с плавающей точкой и одинарной
точностью, тип float
3.14159L   // литерал с плавающей точкой и
расширенной точностью,
// тип long double
```

Рекомендуем

При обозначении литерала как имеющего тип `long` используйте букву `L` в верхнем регистре; строчная буква `l` слишком похожа на цифру 1.

Таблица 2.2. Определение типа литерала

Символьные и строковые литералы			
Префикс	Значение	Тип	
U	Символ Unicode 16	<code>char16_t</code>	
U	Символ Unicode 32	<code>char32_t</code>	
L	Широкий символ	<code>wchar_t</code>	
U8	utf-8 (только строковые литералы)	<code>char</code>	
Целочисленные литералы		Литералы с плавающей точкой	
Суффикс	Минимальный тип	Суффикс	Тип
u или U	<code>unsigned</code>	f или F	<code>float</code>
l или L	<code>long</code>	l или L	<code>long double</code>
Ll или LL	<code>long long</code>		

Можно непосредственно определить знак и размер целочисленного литерала. Если суффикс содержит символ U, то у литерала беззнаковый тип. Таким образом, у десятичного, восьмеричного или

шестнадцатеричного литерала с суффиксом U будет наименьший тип unsigned int, unsigned long или unsigned long long, в соответствии со значением литерала. Если суффикс будет содержать символ L, то типом литерала будет по крайней мере long; если суффикс будет содержать символы LL, то типом литерала будет long long или unsigned long long.

Можно объединить символ U с символом L или символами LL. Литерал с суффиксом UL, например, задаст тип unsigned long или unsigned long long, в зависимости от того, помещается ли его значение в тип unsigned long.

Логические литералы и лiteralные указатели

Слова true и false — это логические литералы (литералы типа bool)

```
bool test = false;
```

Слово nullptr является literalным указателем. Более подробная информация об указателях и литерале nullptr приведена в разделе 2.3.2.

Упражнения раздела 2.1.3

Упражнение 2.5. Определите тип каждого из следующих литералов. Объясните различия между ними:

- (a) 'a', L'a', "a", L"a"
- (b) 10, 10u, 10L, 10uL, 012, 0xC
- (c) 3.14, 3.14f, 3.14L
- (d) 10, 10u, 10., 10e-2

Упражнение 2.6. Имеются ли различия между следующими определениями:

```
int month = 9, day = 7;  
int month = 09, day = 07;
```

Упражнение 2.7. Какие значения представляют эти литералы? Какой тип имеет каждый из них?

- (a) "Who goes with F\145rgus?\012"
- (b) 3.14e1L (c) 1024f (d) 3.14L

Упражнение 2.8. Напишите программу, использующую управляющие последовательности для вывода значения 2M, сопровождаемого новой строкой. Модифицируйте программу так, чтобы вывести 2, затем табуляцию, потом M и наконец символ новой строки.

2.2. Переменные

Переменная (variable) — это именованное хранилище, которым могут манипулировать программы. У каждой переменной в языке C++ есть тип. Тип определяет размер и расположение переменной в памяти, диапазон значений, которые могут храниться в ней, и набор применимых к переменной операций. Программисты C++ используют термины "переменная" и "объект" как синонимы.



2.2.1. Определения переменных

Простое определение переменной состоит из *спецификатора типа* (type specifier), сопровождаемого списком из одного или нескольких имен переменных, отделенных запятыми, и завершающей точки с запятой. Тип каждого имени в списке задан спецификатором типа. Определение может (не обязательно) предоставить исходное значение для одного или нескольких определяемых имен:

```
int sum = 0, value, // sum, value и units_sold
имеют тип int
units_sold = 0; // sum и units_sold
инициализированы значением 0
Sales_item item; // item имеет тип Sales_item
(см. р. 1.5.1)
// string — библиотечный тип, представляющий
последовательность
// символов переменной длины
std::string book("0-201-78345-X"); // book
инициализирована строковым
// литералом
```

В определении переменной `book` использован библиотечный тип `std::string`. Подобно классу `iostream` (см. раздел 1.2), класс `string` определен в пространстве имен `std`. Более подробная информация о классе `string` приведена в главе 3, а пока достаточно знать то, что тип `string` представляет последовательность символов переменной длины. Библиотечный тип `string` предоставляет несколько способов

инициализации строковых объектов. Один из них — копирование строкового литерала (см. раздел 2.1.3). Таким образом, переменная `book` инициализируется символами 0-201-78345-X.

Терминология. Что такое объект?

Программисты языка C++ используют термин *объект* (*object*) часто, и не всегда по делу. В самом общем определении объект — это область памяти, способная содержать данный и обладающая типом.

Одни программисты используют термин *объект* лишь для переменных и экземпляров классов. Другие используют его, чтобы различать именованные и неименованные объекты, причем для именованных объектов используют термин *переменная* (*variable*). Третьи различают объекты и значения, используя термин *объект* для тех данных, которые могут быть изменены программой, и термин *значение* (*value*) — для тех данных, которые предназначены только для чтения.

В этой книге используется наиболее общий смысл термина *объект*, т.е. область памяти, для которой указан тип. Здесь под объектом подразумеваются практически все используемые в программе данные, независимо от того, имеют ли они встроенный тип или тип класса, являются ли они именованными или нет, предназначены только для чтения или допускают изменение.

Инициализаторы

Инициализация (*initialization*) присваивает объекту определенное значение в момент его создания. Используемые для инициализации переменных значения могут быть насколько угодно сложными выражениями. Когда определяется несколько переменных, имена всех объектов следуют непосредственно друг за другом. Таким образом, вполне возможно инициализировать переменную значением одной из переменных, определенных ранее в том же определении.

```
// ok: переменная price определяется и
инициализируется прежде,
// чем она будет использована для инициализации
переменной discount
double price = 109.99, discount = price * 0.16;
// ok: Вызов функции applyDiscount() и
использование ее возвращаемого
// значения для инициализации переменной salePrice
salePrice = applyDiscount(price, discount);
```

Инициализация в C++ — на удивление сложная тема, и мы еще не раз вернемся к ней. Многих программистов вводит в заблуждение использование символа = при инициализации переменной. Они полагают, что инициализация — это такая форма присвоения, но в C++ инициализация и присвоение — совершенно разные операции. Эта концепция особенно важна, поскольку во многих языках это различие несущественно и может быть проигнорировано. Тем не менее даже в языке C++ это различие зачастую не имеет значения. Однако данная концепция крайне важна, и мы будем повторять это еще не раз.



Инициализация — это не присвоение. Инициализация переменной происходит при ее создании. Присвоение удаляет текущее значение объекта и заменяет его новым.

Списочная инициализация

Тема инициализации настолько сложна потому, что язык поддерживает ее в нескольких разных формах. Например, для определения переменной `units_sold` типа `int` и ее инициализации значением 0 можно использовать любой из следующих четырех способов:

```
int units_sold = 0;  
int units_sold = { 0 } ;  
int units_sold{ 0 } ;  
int units_sold( 0 ) ;
```



Использование фигурных скобок для инициализации было введено новым стандартом. Ранее эта форма инициализации допускалась лишь в некоторых случаях. По причинам, описанным в разделе 3.3.1, эта форма инициализации известна как *списочная инициализация* (list initialization). Списки инициализаторов в скобках можно теперь использовать всегда, когда инициализируется объект, и в некоторых случаях, когда объекту присваивается новое значение.

При использовании с переменными встроенного типа эта форма инициализации обладает важным преимуществом: компилятор не позволит инициализировать переменные встроенного типа, если инициализатор

может привести к потере информации:

```
long double ld = 3.1415926536;
int a{ld}, b = {ld}; // ошибка: преобразование с
потерей
int c(ld), d = ld; // ok: но значение будет
усечено
```

Компилятор откажет в инициализации переменных `a` и `b`, поскольку использование значения типа `long double` для инициализации переменной типа `int` может привести к потере данных. Как минимум, дробная часть значения переменной `ld` будет усечена. Кроме того, целочисленная часть значения переменной `ld` может быть слишком большой, чтобы поместиться в переменную типа `int`.

То, что здесь представлено, может показаться тривиальным, в конце концов, вряд ли кто инициализирует переменную типа `int` значением типа `long double` непосредственно. Однако, как представлено в главе 16, такая инициализация может произойти непреднамеренно. Более подробная информация об этих формах инициализации приведена в разделах 3.2.1 и 3.3.1.

Инициализация по умолчанию

При определении переменной без инициализатора происходит ее *инициализация по умолчанию* (default initialization). Таким переменным присваивается *значение по умолчанию* (default value). Это значение зависит от типа переменной и может также зависеть от того, где определяется переменная.

Значение объекта встроенного типа, не инициализированного явно, зависит от того, где именно он определяется. Переменные, определенные вне тела функции, инициализируются значением 0. За одним рассматриваемым вскоре исключением, определенные в функции переменные встроенного типа остаются *неинициализированными* (uninitialized). Значение неинициализированной переменной встроенного типа неопределено (см. раздел 2.1.2). Попытка копирования или получения доступа к значению неинициализированной переменной является ошибкой.

Инициализацию объекта типа класса контролирует сам класс. В частности, класс позволяет определить, могут ли быть созданы его объекты без инициализатора. Если это возможно, класс определяет значение, которое будет иметь его объект в таком случае.

Большинство классов позволяет определять объекты без явных инициализаторов. Такие классы самостоятельно предоставляют соответствующее значение по умолчанию. Например, новый объект библиотечного класса `string` без инициализатора является пустой строкой.

```
std::string empty; // неявно инициализируется  
пустой строкой  
Sales_item item; // объект Sales_item  
инициализируется // значением по умолчанию
```

Однако некоторые классы требуют, чтобы каждый объект был инициализирован явно. При попытке создать объект такого класса без инициализатора компилятор пожалуется на это.



Значение неинициализированных объектов встроенного типа, определенных в теле функции, неопределено. Значение неинициализируемых явно объектов типа класса определяется классом.

Упражнения раздела 2.2.1

Упражнение 2.9. Объясните следующие определения. Если среди них есть некорректные, объясните, что не так и как это исправить.

- (a) `std::cin >> int input_value;` (b) `int i = {3.14};`
- (c) `double salary = wage = 9999.99;` (d) `int i = 3.14;`

Упражнение 2.10. Каковы исходные значения, если таковые вообще имеются, каждой из следующих переменных?

```
std::string global_str;  
int global_int;  
int main()  
{  
    int local_int;  
    std::string local_str;  
}
```



2.2.2. Объявления и определения переменных

Для обеспечения возможности разделить программу на несколько логических частей язык C++ предоставляет технологию, известную как *раздельная компиляция* (*separate compilation*). Раздельная компиляция позволяет составлять программу из нескольких файлов, каждый из которых может быть откомпилирован независимо.

При разделении программы на несколько файлов необходим способ совместного использования кода этих файлов. Например, код, определенный в одном файле, возможно, должен использовать переменную, определенную в другом файле. В качестве конкретного примера рассмотрим объекты `std::cout` и `std::cin`. Классы этих объектов определены где-то в стандартной библиотеке, но все же наши программы могут использовать их.

Внимание! Неинициализированные переменные — причина проблем во время выполнения

Значение неинициализированной переменной неопределено. Попытка использования значения неинициализированной переменной является ошибкой, которую зачастую трудно обнаружить. Кроме того, компилятор не обязан обнаруживать такие ошибки, хотя большинство из них предупреждает, по крайней мере, о некоторых случаях использования неинициализированных переменных.

Что же произойдет при использовании неинициализированной переменной с неопределенным значением? Иногда (если повезет) программа отказывает сразу, при попытке доступа к объекту. Обнаружив место, где происходит отказ, как правило, довольно просто выяснить, что его причиной является неправильно инициализированная переменная. Но иногда программа срабатывает, хотя результат получается ошибочным. Возможен даже худший вариант, когда на одной машине результаты получаются правильными, а на другой происходит сбой. Кроме того, добавление кода во вполне работоспособную программу в неподходящем месте тоже может привести к внезапному возникновению проблем.



ВНИМАНИЕ

Мы рекомендуем инициализировать каждый объект встроенного типа. Это не всегда необходимо, но проще и безопасней предоставить инициализатор, чем выяснить, можно ли в данном конкретном случае безопасно опустить его.

Для поддержки раздельной компиляции язык C++ различает объявления и определения. *Объявление* (declaration) делает имя известным программе. Файл, который должен использовать имя, определенное в другом месте, включает объявление для этого имени. *Определение* (definition) создает соответствующую сущность.

Объявление переменной определяет ее тип и имя. Определение переменной — это ее объявление. Кроме задания имени и типа, определение резервирует также место для ее хранения и может снабдить переменную исходным значением.

Чтобы получить объявление, не являющееся также определением, добавляется ключевое слово `extern` и можно не предоставлять явный инициализатор.

```
extern int i; // объявить, но не определить  
переменную i
```

```
int j; // объявить и определить переменную j
```

Любое объявление, которое включает явный инициализатор, является определением. Для переменной, определенной как `extern` (внешняя), можно предоставить инициализатор, но это отменит ее определение как `extern`. Объявление внешней переменной с инициализатором является ее определением:

```
extern double pi = 3.1416; // определение
```

Предоставление инициализатора внешней переменной в функции является ошибкой.



Объявлены переменные могут быть много раз, но определены только однажды.

На настоящий момент различие между объявлением и определением может показаться неочевидным, но фактически оно очень важно. Использование переменной в нескольких файлах требует объявлений, отдельных от определения. Чтобы использовать ту же переменную в нескольких файлах, ее следует определить в одном, и только одном файле. В других файлах, где используется та же переменная, ее следует объявить, но не определять.

Более подробная информация о том, как язык C++ поддерживает раздельную компиляцию, приведена в разделах 2.6.3 и 6.1.3.

Упражнения раздела 2.2.2

Упражнение 2.11. Объясните, приведены ли ниже объявления или определения.

- (a) `extern int ix = 1024;`
- (b) `int iy;`
- (c) `extern int iz;`

Ключевая концепция. Статическая типизация

Язык C++ обладает строгим *статическим контролем типов* (statically typed) данных. Это значит, что проверка соответствия значений заявленным для них типам данных осуществляется во время компиляции. Сам процесс проверки называют *контролем соответствия типов* (type-checking), или *типовацией* (typing).

Как уже упоминалось, тип ограничивает операции, которые можно выполнять с объектом. В языке C++ компилятор проверяет, поддерживает ли используемый тип операции, которые с ним выполняют. Если обнаруживается попытка сделать нечто, не поддерживаемое данным типом, компилятор выдает сообщение об ошибке и не создает исполняемый файл.

По мере усложнения рассматриваемых программ будет со всей очевидностью продемонстрировано, что строгий контроль соответствия типов способен помочь при поиске ошибок в исходном коде. Однако последствием статической проверки является то, что тип каждой используемой сущности должен быть известен компилятору. Следовательно, тип переменной необходимо объявить прежде, чем эту переменную можно будет использовать.

2.2.3. Идентификаторы

Идентификаторы (*identifier*) (или имена) в языке C++ могут состоять из символов, цифр и символов подчеркивания. Язык не налагает ограничений на длину имен. Идентификаторы должны начинаться с букв или символа подчеркивания. Символы в верхнем и нижнем регистрах различаются, т.е. идентификаторы языка C++ чувствительны к регистру.

```
// определено четыре разных переменных типа int
int somename, someName, SomeName, SOMENAME;
```

Язык резервирует набор имен, перечисленных в табл. 2.3 и 2.4, для собственных нужд. Эти имена не могут использоваться как идентификаторы.

Таблица 2.3. Ключевые слова языка C++

alignas	continue	friend	register	true
alignof	decltype	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	do	int	signed	typename
break	double	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert	using
char	enum	new	static_cast	virtual
char16_t	explicit	noexcept	struct	void
char32_t	export	nullptr	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr	float	protected	thread_local	
const_cast	for	public	throw	

Таблица 2.4. Альтернативные имена операторов языка C++

and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bitor	not	or	xor	

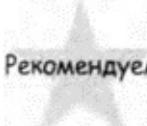
Кроме ключевых слов, стандарт резервирует также набор идентификаторов для использования в библиотеке, поэтому пользовательские идентификаторы не могут содержать два последовательных символа подчеркивания, а также начинаться с символа подчеркивания, непосредственно за которым следует прописная буква.

Кроме того, идентификаторы, определенные вне функций, не могут начинаться с символа подчеркивания.

Соглашения об именах переменных

Существует множество общепринятых соглашений для именования переменных. Применение подобных соглашений может существенно улучшать удобочитаемость кода.

- Идентификатор должен быть осмысленным.
- Имена переменных обычно состоят из строчных символов. Например, `index`, а не `Index` или `INDEX`.
- Имена классов обычно начинаются с прописной буквы, например `Sales_item`.
- Несколько слов в идентификаторе разделяют либо символом подчеркивания, либо прописными буквами в первых символах каждого слова. Например: `student_loan` или `studentLoan`, но не `studentloan`.



Рекомендуем

Самым важным аспектом соглашения об именовании является его неукоснительное соблюдение.

Упражнения раздела 2.2.3

Упражнение 2.12. Какие из приведенных ниже имен недопустимы (если таковые есть)?

- (a) `int double = 3.14;` (b) `int _;`
- (c) `int catch-22;` (d) `int 1_or_2 = 1;`
- (e) `double Double = 3.14;`

2.2.4. Область видимости имен

В любом месте программы каждое используемое имя относится к вполне определенной сущности — переменной, функции, типу и т.д. Однако имя может быть использовано многоократно для обращения к различным сущностям в разных точках программы.

Область видимости (*scope*) — это часть программы, в которой у имени есть конкретное значение. Как правило, области видимости в языке C++ разграничиваются фигурными скобками.

В разных областях видимости то же имя может относиться к разным сущностям. Имена видимы от момента их объявления и до конца области видимости, в которой они объявлены.

В качестве примера рассмотрим программу из раздела 1.4.2:

```
#include <iostream>
int main() {
    int sum = 0;
    // сложить числа от 1 до 10 включительно
    for (int val = 1; val <= 10; ++val)
        sum += val; // эквивалентно sum = sum + val
    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

Эта программа определяет три имени — `main`, `sum` и `val`, а также использует имя пространства имен `std`, наряду с двумя именами из этого пространства имен — `cout` и `endl`.

Имя `main` определено вне фигурных скобок. Оно, как и большинство имен, определенных вне функции, имеет *глобальную область видимости* (global scope). Будучи объявлены, имена в глобальной области видимости доступны в программе повсюду. Имя `sum` определено в пределах блока, которым является тело функции `main()`. Оно доступно от момента объявления и далее в остальной части функции `main()`, но не за ее пределами. Переменная `sum` имеет *область видимости блока* (block scope). Имя `val` определяется в пределах оператора `for`. Оно применимо только в этом операторе, но не в другом месте функции `main()`.

Совет. Определяйте переменные при первом использовании

Объект имеет смысл определять поближе к месту его первого использования. Это улучшает удобочитаемость и облегчает поиск определения переменной. Однако важней всего то, что когда переменная определяется ближе к месту ее первого использования, зачастую проще присвоить ей подходящее исходное значение.

Вложенные области видимости

Области видимости могут содержать другие области видимости. Содержащаяся (или вложенная) область видимости называется *внутренней областью видимости* (inner scope), а содержащая ее область видимости —

внешней областью видимости (outer scope).

Как только имя объявлено в области видимости, оно становится доступно во вложенных в нее областях видимости. Имена, объявленные во внешней области видимости, могут быть также переопределены во внутренней области видимости:

```
#include <iostream>
// Программа предназначена исключительно для демонстрации.
// Использование в функции глобальной переменной, а также определение
// одноименной локальной переменной - это очень плохой стиль
// программирования
int reused = 42; // reused имеет глобальную область видимости
int main()
{
    int unique = 0; // unique имеет область видимости блока
    // вывод #1: используется глобальная reused;
    // выводит 42 0
    std::cout << reused << " " << unique << std::endl;
    int reused = 0; // новый локальный объект по имени reused скрывает
                    // глобальный reused
    // вывод #2: используется локальная reused;
    // выводит 0 0
    std::cout << reused << " " << unique << std::endl;
    // вывод #3: явное обращение к глобальной reused;
    // выводит 42 0
    std::cout << ::reused << " " << unique <<
std::endl;
    return 0;
}
```

Вывод #1 осуществляется перед определением локальной переменной reused. Поэтому данный оператор вывода использует имя reused, определенное в глобальной области видимости. Этот оператор выводит 42 0. Вывод #2 происходит после определения локальной переменной

`reused`. Теперь локальная переменная `reused` находится в области видимости (in scope). Таким образом, второй оператор вывода использует локальный объект `reused`, а не глобальный и выводит 0 0. Вывод #3 использует оператор области видимости (см. раздел 1.2) для переопределения стандартных правил областей видимости. У глобальной области видимости нет имени. Следовательно, когда у оператора области видимости пусто слева, это обращение к указанному справа имени в глобальной области видимости. Таким образом, это выражение использует глобальный объект `reused` и выводит 42 0.



Как правило, определение локальных переменных, имена которых совпадают с именами глобальных переменных, является крайне неудачным решением.

Упражнения раздела 2.2.4

Упражнение 2.13. Каково значение переменной `j` в следующей программе?

```
int i = 42;
int main() {
    int i = 100;
    int j = i;
}
```

Упражнение 2.14. Допустим ли следующий код? Если да, то какие значения он отобразит на экране?

```
int i = 100, sum = 0;
for (int i = 0; i != 10; ++i)
    sum += i;
std::cout << i << " " << sum << std::endl;
```



2.3. Составные типы

Составной тип (compound type) — это тип, определенный в терминах другого типа. У языка C++ есть несколько составных типов, два из которых, ссылки и указатели, мы рассмотрим в этой главе.

У рассмотренных на настоящий момент объявлений не было ничего, кроме имен переменных. Такие переменные имели простейший, базовый тип объявления. Более сложные операторы объявления позволяют определять переменные с составными типами, которые состоят из объявлений базового типа.



2.3.1. Ссылки

Ссылка (reference) является альтернативным именем объекта. Ссылочный тип "ссылается на" другой тип. В определении ссылочного типа используется оператор объявления в форме `&d`, где `d` — объявляемое имя:

```
int ival = 1024;  
int &refVal = ival; // refVal ссылается на другое  
имя, ival  
int &refVal2; // ошибка: ссылку следует  
инициализировать
```

Обычно при инициализации переменной значение инициализатора копируется в создаваемый объект. При определении ссылки вместо копирования значения инициализатора происходит *связывание* (bind) ссылки с ее инициализатором. После инициализации ссылка остается связанной с исходным объектом. Нет никакого способа изменить привязку ссылки так, чтобы она ссылалась на другой объект, поэтому ссылки *следует инициализировать*.



Новый стандарт ввел новый вид ссылки — *ссылка r-значения* (r-value reference), которую мы рассмотрим в разделе 13.6.1. Эти ссылки предназначены прежде всего для использования в классах. С технической точки зрения, когда мы используем термин ссылка (reference), мы подразумеваем *ссылку l-значения* (l-value reference).



Ссылка — это псевдоним

После того как ссылка определена, *все* операции с ней фактически осуществляются с объектом, с которым связана ссылка.

`refVal = 2; // присваивает значение 2 объекту, на который ссылается`

`// ссылка refVal, т. е. ival
int ii = refVal; // то же, что и ii = ival`



Ссылка — это не объект, а *только другое имя уже существующего объекта*.

При присвоении ссылки присваивается объект, с которым она связана. При доступе к значению ссылки фактически происходит обращение к значению объекта, с которым связана ссылка. Точно так же, когда ссылка используется как инициализатор, в действительности для этого используется объект, с которым связана ссылка.

`// ok: ссылка refVal3 связывается с объектом, с которым связана
// ссылка refVal, т. е. с ival
int &refVal3 = refVal;
// инициализирует i значением объекта, с которым
// связана ссылка refVal
int i = refVal; // ok: инициализирует i значением ival`

Поскольку ссылки не объекты, нельзя определить ссылку на ссылку.

Определение ссылок

В одном определении можно определить несколько ссылок. Каждому являемомуся ссылкой идентификатору должен предшествовать символ &.

```
int i = 1024, i2 = 2048; // i и i2 - переменные
типа int
int &r = i, r2 = i2;      // r - ссылка, связанная с
переменной i;
                           // r2 - переменная типа
int
int i3 = 1024, &ri = i3; // i3 - переменная типа
int;                      // ri - ссылка, связанная
с переменной i3
int &r3 = i3, &r4 = i2; // r3 и r4 - ссылки
```

За двумя исключениями, рассматриваемыми в разделах 2.4.1 и 15.2.3, типы ссылки и объекта, на который она ссылается, должны совпадать точно. Кроме того, по причинам, рассматриваемым в разделе 2.4.1, ссылка может быть связана только с объектом, но не с литералом или результатом более общего выражения:

```
int &refVal4 = 10;        // ошибка: инициализатор
должен быть объектом
double dval = 3.14;
int &refVal5 = dval;    // ошибка: инициализатор
должен быть объектом
                           // типа int
```

Упражнения раздела 2.3.1

Упражнение 2.15. Какие из следующих определений недопустимы (если таковые есть)? Почему?

- (a) int ival = 1.01; (b) int &rval1 = 1.01;
- (c) int &rval2 = ival; (d) int &rval3;

Упражнение 2.16. Какие из следующих присвоений недопустимы (если таковые есть)? Если они допустимы, объясните, что они делают.

- int i = 0, &r1 = i; double d = 0, &r2 = d;
- (a) r2 = 3.14159; (b) r2 = r1;
- (c) i = r2; (d) r1 = d;

Упражнение 2.17. Что выводит следующий код?

```
int i, &ri = i;
i = 5; ri = 10;
std::cout << i << " " << ri << std::endl;
```



2.3.2. Указатели

Указатель (pointer) — это составной тип, переменная которого указывает на объект другого типа. Подобно ссылкам, указатели используются для косвенного доступа к другим объектам. В отличие от ссылок, указатель — это настоящий объект. Указатели могут быть присвоены и скопированы; один указатель за время своего существования может указывать на несколько разных объектов. В отличие от ссылки, указатель можно не инициализировать в момент определения. Подобно объектам других встроенных типов, значение неинициализированного указателя, определенного в области видимости блока, неопределено.



ВНИМАНИЕ

Указатели зачастую трудно понять. При отладке проблемы, связанные с ошибками в указателях, способны запутать даже опытных программистов.

Тип указателя определяется оператором в форме `*d`, где `d` — определяемое имя. Символ `*` следует повторять для каждой переменной указателя.

```
int *ip1, *ip2; // ip1 и ip2 — указатели на тип int
double dp, *dp2; // dp2 — указатель на тип double;
                  // dp — переменная типа double
```

Получение адреса объекта

Указатель содержит адрес другого объекта. Для получения адреса объекта используется *оператор обращения к адресу* (address-of operator), или *оператор &*.

```
int ival = 42;
int *p = &ival; // p содержит адрес переменной ival
                // p — указатель на переменную ival
```

Второй оператор определяет `p` как указатель на тип `int` и инициализирует его адресом объекта `ival` типа `int`. Поскольку ссылки не

объекты, у них нет адресов, а следовательно, невозможно определить указатель на ссылку.

За двумя исключениями, рассматриваемыми в разделах 2.4.2 и 15.2.3, типы указателя и объекта, на который он указывает, должны совпадать.

```
double dval;
double *pd = &dval; // ok: инициализатор - адрес
объекта типа double
double *pd2 = pd; // ok: инициализатор -
указатель на тип double
int *pi = pd; // ошибка: типы pi и pd
отличаются
pi = &dval; // ошибка: присвоение адреса
типа double
// указателю на тип int
```

Типы должны совпадать, поскольку тип указателя используется для выводения типа объекта, на который он указывает. Если бы указатель содержал адрес объекта другого типа, то выполнение операций с основным объектом потерпело бы неудачу.

Значение указателя

Хранимое в указателе значение (т.е. адрес) может находиться в одном из четырех состояний.

1. Оно может указывать на объект.
2. Оно может указывать на область непосредственно за концом объекта
3. Это может быть нулевое значение, означающее, что данный указатель не связан ни с одним объектом.
4. Оно может быть недопустимо. Любое иное значение, кроме приведенного выше, является недопустимым.

Копирование или иная попытка доступа к значению по недопустимому указателю является серьезной ошибкой. Как и использование неинициализированной переменной, компилятор вряд ли обнаружит эту ошибку. Результат доступа к недопустимому указателю непредсказуем. Поэтому всегда следует знать, допустим ли данный указатель.

Хотя указатели в случаях 2 и 3 допустимы, действия с ними ограничены. Поскольку эти указатели не указывают ни на какой объект, их нельзя использовать для доступа к объекту. Если все же сделать попытку доступа к объекту по такому указателю, то результат будет непредсказуем.

Использование указателя для доступа к объекту

Когда указатель указывает на объект, для доступа к этому объекту можно использовать *оператор обращения к значению* (dereference operator), или *оператор **.

```
int ival = 42;
int *p = &ival; // p содержит адрес ival; p -
указатель на ival
cout << *p; // * возвращает объект, на который
указывает p;
// выводит 42
```

Обращение к значению указателя возвращает объект, на который указывает указатель. Присвоив значение результату оператора обращения к значению, можно присвоить его самому объекту.

```
*p = 0; // * возвращает объект; присвоение
нового значения
// ival через указатель p
cout << *p; // выводит 0
```

При присвоении значения *p оно присваивается объекту, на который указывает указатель p.



Обратиться к значению можно только по допустимому указателю, который указывает на объект.

Ключевая концепция. У некоторых символов есть несколько значений

Некоторые символы, такие как & и *, используются и как оператор в выражении, и как часть объявления. Контекст, в котором используется символ, определяет то, что он означает.

```
int i = 42;
int &r = i; // & следует за типом в части
объявления; r - ссылка
int *p; // * следует за типом в части
объявления; p - указатель
p = &i; // & используется в выражении как
оператор
// обращения к адресу
```

```
*p = i;           // * используется в выражении как
оператор          // обращения к значению
int &r2 = *p; // & в части объявления; * -
оператор обращения к значению
```

В объявлениях символы `&` и `*` используются для формирования составных типов. В выражениях эти же символы используются для обозначения оператора. Поскольку тот же символ используется в совершенно ином смысле, возможно, стоит игнорировать внешнее сходство и считать их как будто различными символами.

Нулевые указатели

Нулевой указатель (null pointer) не указывает ни на какой объект. Код может проверить, не является ли указатель нулевым, прежде чем пытаться использовать его. Есть несколько способов получить нулевой указатель.

```
int *p1 = nullptr; // эквивалентно int *p1 = 0;
int *p2 = 0;           // непосредственно
инициализирует p2 литеральной
                           // константой 0, необходимо
#include <cstdlib>
int *p3 = NULL;      // эквивалентно int *p3 = 0;
```



Проще всего инициализировать указатель, используя *литерал* `nullptr`, который был введен новым стандартом. Литерал `nullptr` имеет специальный тип, который может быть преобразован (см. раздел 2.1.2) в любой другой ссылочный тип. В качестве альтернативы можно инициализировать указатель литералом `0`, как это сделано в определении указателя `p2`.

Программисты со стажем иногда используют *переменную препроцессора* (preprocessor variable) `NULL`, которую заголовок `cstdlib` определяет как `0`.

Немного подробней препроцессор рассматривается в разделе 2.6.3, а пока достаточно знать, что *препроцессор* (preprocessor) — это программа, которая выполняется перед компилятором. Переменные препроцессора используются препроцессором, они не являются частью пространства имен `std`, поэтому их указывают непосредственно, без префикса `std::`.

При использовании переменной препроцессора последний

автоматически заменяет такую переменную ее значением. Следовательно, инициализация указателя переменной `NULL` эквивалентна его инициализации значением `0`. Сейчас программы C++ вообще должны избегать применения переменной `NULL` и использовать вместо нее литерал `nullptr`.

Нельзя присваивать переменную типа `int` указателю, даже если ее значением является `0`.

```
int zero = 0;  
pi = zero; // ошибка: нельзя присвоить переменную  
типа int указателю
```

Совет. Инициализируйте все указатели

Неинициализированные указатели — обычный источник ошибок времени выполнения.

Подобно любой другой неинициализированной переменной, последствия использования неинициализированного указателя непредсказуемы. Использование неинициализированного указателя почти всегда приводит к аварийному отказу во время выполнения. Однако поиск причин таких отказов может оказаться на удивление трудным.

У большинства компиляторов при использовании неинициализированного указателя биты в памяти, где он располагается, используются как адрес. Использование неинициализированного указателя — это попытка доступа к несуществующему объекту в произвольной области памяти. Нет никакого способа отличить допустимый адрес от недопустимого, состоящего из случайных битов, находящихся в той области памяти, которая была зарезервирована для указателя.

Авторы рекомендуют инициализировать все переменные, а особенно указатели. Если это возможно, определяйте указатель только после определения объекта, на который он должен указывать. Если связываемого с указателем объекта еще нет, то инициализируйте указатель значением `nullptr` или `0`. Так код программы может узнать, что указатель не указывает на объект.

Присвоение и указатели

И указатели, и ссылки предоставляют косвенный доступ к другим объектам. Однако есть важные различия в способе, которым они это делают. Самое важное то, что ссылка — это не объект. После того как

ссылка определена, нет никакого способа заставить ее ссылаться на другой объект. При использовании ссылки всегда используется объект, с которым она была связана первоначально.

Между указателем и содержащимся в нем адресом нет такой связи. Подобно любой другой (нессылочной) переменной, при присвоении указателя для него устанавливается новое значение. Присвоение заставляет указатель указывать на другой объект.

```
int i = 42;
int *pi = 0;      // указатель pi инициализирован, но
не адресом объекта
int *pi2 = &i;    // указатель pi2 инициализирован
адресом объекта i
int *pi3;         // если pi3 определен в блоке, pi3
не инициализирован
pi3 = pi2;        // pi3 и pi2 указывают на тот же
объект, т. е. на i
pi2 = 0;          // теперь pi2 не содержит адреса
никакого объекта
```

Сначала может быть трудно понять, изменяет ли присвоение указатель или сам объект, на который он указывает. Важно не забывать, что присвоение изменяет свой левый операнд. Следующий код присваивает новое значение переменной *pi*, что изменяет адрес, который она хранит:

```
pi = &ival; // значение pi изменено; теперь pi
указывает на ival
```

С другой стороны, следующий код (использующий **pi*, т.е. значение, на которое указывает указатель *pi*) изменяет значение объекта:

```
*pi = 0; // значение ival изменено; pi неизменен
```

Другие операции с указателями

Пока значение указателя допустимо, его можно использовать в условии. Аналогично использованию арифметических значений (раздел 2.1.2), если указатель содержит значение 0, то условие считается ложным.

```
int ival = 1024;
int *pi = 0;      // pi допустим, нулевой указатель
int *pi2 = &ival; // pi2 допустим, содержит адрес
ival
if (pi)           // pi содержит значение 0,
условие считается ложным
// ...
```

```
if ( pi2 ) // pi2 указывает на ival, значит,  
содержит не 0; // условие считается истинным  
// ...
```

Любой отличный от нулевого указатель рассматривается как значение `true`. Два допустимых указателя того же типа можно сравнить, используя операторы равенства (`==`) и неравенства (`!=`). Результат этих операторов имеет тип `bool`. Два указателя равны, если они содержат одинаковый адрес, и неравны в противном случае. Два указателя содержат одинаковый адрес (т.е. равны), если они оба нулевые, если они указывают на тот же объект или на область непосредственно за концом того же объекта. Обратите внимание, что указатель на объект и указатель на область за концом другого объекта вполне могут содержать одинаковый адрес. Такие указатели равны.

Поскольку операции сравнения используют значения указателей, эти указатели должны быть допустимы. Результат использования недопустимого указателя в условии или в сравнении непредсказуем.

Дополнительные операции с указателями будут описаны в разделе 3.5.3.

Тип `void*` является специальным типом указателя, способного содержать адрес любого объекта. Подобно любому другому указателю, указатель `void*` содержит адрес, но тип объекта по этому адресу неизвестен.

```
double obj = 3.14, *pd = &obj;  
// ok: void* может содержать адрес любого типа  
данных  
void *pv = &obj; // obj может быть объектом любого  
типа  
pv = pd; // pv может содержать указатель на  
любой тип
```

С указателем `void*` допустимо немного действий: его можно сравнить с другим указателем, можно передать его функции или возвратить из нее либо присвоить другому указателю типа `void*`. Его нельзя использовать для работы с объектом, адрес которого он содержит, поскольку неизвестен тип объекта, неизвестны и операции, которые можно с ним выполнять.

Как правило, указатель `void*` используют для работы с памятью как с областью памяти, а не для доступа к объекту, хранящемуся в этой области. Использование указателей `void*` рассматривается в разделе 19.1.1, а в

разделе 4.11.3 продемонстрировано, как можно получить адрес, хранящийся в указателе `void*`.

Упражнения раздела 2.3.2

Упражнение 2.18. Напишите код, изменяющий значение указателя. Напишите код для изменения значения, на которое указывает указатель.

Упражнение 2.19. Объясните основные отличия между указателями и ссылками.

Упражнение 2.20. Что делает следующая программа?

```
int i = 42;
int *p1 = &i;
*p1 = *p1 * *p1;
```

Упражнение 2.21. Объясните каждое из следующих определений. Укажите, все ли они корректны и почему.

```
int i = 0;
(a) double* dp = &i; (b) int *ip = i; (c) int *p = &i;
```

Упражнение 2.22. С учетом того, что `p` является указателем на тип `int`, объясните следующий код:

```
if (p) // ...
if (*p) // ...
```

Упражнение 2.23. Есть указатель `p`, можно ли определить, указывает ли он на допустимый объект? Если да, то как? Если нет, то почему?

Упражнение 2.24. Почему инициализация указателя `p` допустима, а указателя `lp` нет?

```
int i = 42; void *p = &i; long *lp = &i;
```



2.3.3. Понятие описаний составных типов

Как уже упоминалось, определение переменной состоит из указания базового типа и списка операторов объявления. Каждый оператор объявления может связать свою переменную с базовым типом отдельно от других операторов объявления в том же определении. Таким образом, одно определение может определять переменные различных типов.

```
// i - переменная типа int; p - указатель на тип
int;
```

```
// r - ссылка на тип int
int i = 1024, *p = &i, &r = i;
```



Многие программисты не понимают взаимодействия базового и модифицированного типа, который может быть частью оператора объявления.



Определение нескольких переменных

Весьма распространенное заблуждение полагать, что модификатор типа (`*` или `&`) применяется ко всем переменным, определенным в одном операторе. Частично причина в том, что между модификатором типа и объявляемым именем может находиться пробел.

```
int* p; // вполне допустимо, но может ввести в заблуждение
```

Данное определение может ввести в заблуждение потому, что создается впечатление, будто `int*` является типом каждой переменной, объявленной в этом операторе. Несмотря на внешний вид, базовым типом этого объявления является `int`, а не `int*`. Символ `*` — это модификатор типа `p`, он не имеет никакого отношения к любым другим объектам, которые могли бы быть объявлены в том же операторе:

```
int* p1, p2; // p1 - указатель на тип int; p2 - переменная типа int
```

Есть два общепринятых стиля определения нескольких переменных с типом указателя или ссылки. Согласно первому, модификатор типа располагается рядом с идентификатором:

```
int *p1, *p2; // p1 и p2 - указатели на тип int
```

Этот стиль подчеркивает, что переменная имеет составной тип. Согласно второму, модификатор типа располагается рядом с типом, но он определяет только одну переменную в операторе:

```
int* p1; // p1 - указатель на тип int
int* p2; // p2 - указатель на тип int
```

Этот стиль подчеркивает, что объявление определяет составной тип.



Нет никакого единственно правильного способа определения указателей и ссылок. Важно неукоснительно придерживаться выбранного стиля.

В этой книге используется первый стиль, знак * (или &) помещается рядом с именем переменной.

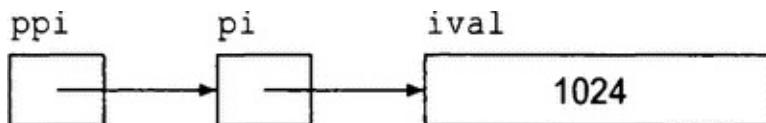
Указатели на указатели

Теоретически нет предела количеству модификаторов типа, применяемых в операторе объявления. Когда модификаторов более одного, они объединяются хоть и логичным, но не всегда очевидным способом. В качестве примера рассмотрим указатель. Указатель — это объект в памяти, и, как у любого объекта, у этого есть адрес. Поэтому можно сохранить адрес указателя в другом указателе.

Каждый уровень указателя отмечается собственным символом *. Таким образом, для указателя на указатель пишут **, для указателя на указатель на указатель — *** и т.д.

```
int ival = 1024;
int *pi = &ival; // pi указывает на переменную типа int
int **ppi = &pi; // ppi указывает на указатель на переменную типа int
```

Здесь *pi* — указатель на переменную типа *int*, а *ppi* — указатель на указатель на переменную типа. Эти объекты можно было бы представить так:



Подобно тому, как обращение к значению указателя на переменную типа *int* возвращает значение типа *int*, обращение к значению указателя на указатель возвращает указатель. Для доступа к основной объекту в этом случае необходимо обратиться к значению указателя дважды:

```
cout << "The value of ival\n"
<< "direct value: " << ival << "\n"
<< "indirect value: " << *pi << "\n"
```

```
<< "doubly indirect value: " << *ppi << endl;
```

Эта программа выводит значение переменной `ival` тремя разными способами: сначала непосредственно, затем через указатель `ri` на тип `int` и наконец обращением к значению указателя `ppi` дважды, чтобы добраться до основного значения в переменной `ival`.

Ссылки на указатели

Ссылка — не объект. Следовательно, не может быть указателя на ссылку. Но поскольку указатель — это объект, вполне можно определить ссылку на указатель.

```
int i = 42;  
int *p;           // p - указатель на тип int  
int *&r = p;    // r - ссылка на указатель p  
r = &i;          // r ссылается на указатель;  
                  // присвоение &i ссылке r делает р  
указателем на i  
*r = 0;           // обращение к значению r дает i,  
объект, на который  
                  // указывает p; изменяет значение i на  
0
```

Проще всего понять тип `r` — прочитать определение справа налево. Ближайший символ к имени переменной (в данном случае `&` в `&r`) непосредственно влияет на тип переменной. Таким образом, становится ясно, что `r` является ссылкой. Остальная часть оператора объявления определяет тип, на который ссылается ссылка `r`. Следующий символ, в данном случае `*`, указывает, что тип `r` относится к типу указателя. И наконец, базовый тип объявления указывает, что `r` — это ссылка на указатель на переменную типа `int`.



Сложное объявление указателя или ссылки может быть проще понять, если читать его справа налево.

Упражнения раздела 2.3.3

Упражнение 2.25. Определите типы и значения каждой из следующих переменных:

(a) int* ip, &r = ip; (b) int i, *ip = 0; (c) int*
ip, ip2;



2.4. Спецификатор `const`

Иногда необходимо определить переменную, значение которой, как известно, не может быть изменено. Например, можно было бы использовать переменную, хранящую размер буфера. Использование переменной облегчит изменение размера буфера, если мы решим, что исходный размер нас не устраивает. С другой стороны, желательно предотвратить непреднамеренное изменение в коде значения этой переменной. Значение этой переменной можно сделать неизменным, используя в ее определении *спецификатор const* (qualifier `const`):

```
const int bufSize = 512; // размер буфера ввода
```

Это определит переменную `bufSize` как константу. Любая попытка присвоить ей значение будет ошибкой:

```
bufSize = 512; // ошибка: попытка записи в константный объект
```

Поскольку нельзя изменить значение константного объекта после создания, его следует инициализировать. Как обычно, инициализатор может быть выражением любой сложности:

```
const int i = get_size(); // ok: инициализация во время выполнения
```

```
const int j = 42; // ok: инициализация во время компиляции
```

```
const int k; // ошибка: k - неинициализированная константа
```

Инициализация и константы

Как уже упоминалось не раз, тип объекта определяет операции, которые можно с ним выполнять. Константный тип можно использовать для большинства, но не для всех операций, как и его неконстантный аналог. Ограничение одно — можно использовать только те операции, которые неспособны изменить объект. Например, тип `const int` можно использовать в арифметических выражениях точно так же, как обычный неконстантный тип `int`. Тип `const int` преобразуется в тип `bool` тем же способом, что и обычный тип `int`, и т.д.

К операциям, не изменяющим значение объекта, относится

инициализация. При использовании объекта для инициализации другого объекта не имеет значения, один или оба из них являются константами.

```
int i = 42;
const int ci = i; // ok: значение i копируется в ci
int j = ci;      // ok: значение ci копируется в j
```

Хотя переменная `ci` имеет тип `const int`, ее значение имеет тип `int`. Константность переменной `ci` имеет значение только для операций, которые могли бы изменить ее значение. При копировании переменной `ci` для инициализации переменной `j` ее константность не имеет значения. Копирование объекта не изменяет его. Как только копия сделана, у нового объекта нет никакой дальнейшей связи с исходным объектом.

По умолчанию константные объекты локальны для файла

Когда константный объект инициализируется константой во время компиляции, такой как `bufSize` в определении ниже, компилятор обычно заменяет используемую переменную ее значением во время компиляции.

```
const int bufSize = 512; // размер буфера ввода
```

Таким образом, компилятор создаст исполняемый код, использующий значение 512 в тех местах, где исходный код использует переменную `bufSize`.

Чтобы заменить переменную значением, компилятор должен видеть ее инициализатор. При разделении программы на несколько файлов, в каждом из которых используется константа, необходим доступ к ее инициализатору. Для этого переменная должна быть определена в каждом файле, в котором используется ее значение (см. раздел 2.2.2). Для обеспечения такого поведения, но все же без повторных определений той же переменной, константные переменные определяются как локальные для файла. Определение константы с тем же именем в нескольких файлах подобно написанию определения для отдельных переменных в каждом файле.

Иногда константу необходимо совместно использовать в нескольких файлах, однако ее инициализатор не является константным выражением. Мы не хотим, чтобы компилятор создал отдельную переменную в каждом файле, константный объект должен вести себя как другие (не константные) переменные. В таком случае определить константу следует в одном файле, и объявить ее в других файлах, где она тоже используется.

Для определения единого экземпляра константной переменной используется ключевое слово `extern` как в ее определении, так и в ее

объявлениях.

```
// файл file_1.cc. Определение и инициализация
константы, которая
// доступна для других файлов
extern const int bufSize = fcn();
// файл file_1.h
extern const int bufSize; // та же bufSize,
определенная в file_1.cc
```

Здесь переменная `bufSize` определяется и инициализируется в файле `file_1.cc`. Поскольку это объявление включает инициализатор, оно (как обычно) является и определением. Но поскольку `bufSize` константа, необходимо применить ключевое слово `extern`, чтобы использовать ее в других файлах.

Объявление в заголовке `file_1.h` также использует ключевое слово `extern`. В данном случае это демонстрирует, что имя `bufSize` не является локальным для этого файла и что его определение находится в другом месте.



Чтобы совместно использовать константный объект в нескольких файлах, его необходимо определить с использованием ключевого слова `extern`.

Упражнения раздела 2.4

Упражнение 2.26. Что из приведенного ниже допустимо? Если что-то недопустимо, то почему?

- (a) `const int buf;`
- (b) `int cnt = 0;`
- (c) `const int sz = cnt;`
- (d) `++cnt; ++sz;`



2.4.1. Ссылка на константу

Подобно любым другим объектам, с константным объектом можно связать ссылку. Для этого используется *ссылка на константу* (reference to `const`), т.е. ссылка на объект типа `const`. В отличие от обычной ссылки, ссылку на константу нельзя использовать для изменения объекта, с

которым она связана.

```
const int ci = 1024;
const int &r1 = ci; // ok: и ссылка, и основной
объект — константы
r1 = 42;           // ошибка: r1 — ссылка на
константу
int &r2 = ci; // ошибка: неконстантная ссылка на
константный объект
```

Поскольку нельзя присвоить значение самой переменной `ci`, ссылка также не должна позволять изменять ее. Поэтому инициализация ссылки `r2` — это ошибка. Если бы эта инициализация была допустима, то ссылку `r2` можно было бы использовать для изменения значения ее основного объекта.

Терминология. Константная ссылка — это ссылка на константу

Программисты C++, как правило, используют термин *константная ссылка* (`const reference`), однако фактически речь идет о *ссылке на константу* (`reference to const`).

С технической точки зрения нет никаких константных ссылок. Ссылка — не объект, поэтому саму ссылку нельзя сделать константой. На самом деле, поскольку нет никакого способа заставить ссылку ссылаться на другой объект, то в некотором смысле все ссылки — константы. То, что ссылка ссылается на константный или неконстантный тип, относится к тому, что при помощи этой ссылки можно сделать, однако привязку самой ссылки изменить нельзя в любом случае.

Инициализация и ссылки на константу

В разделе 2.1.2 мы обращали ваше внимание на два исключения из правила, согласно которому тип ссылки должен совпадать с типом объекта, на который она ссылается. Первое исключение: мы можем инициализировать ссылку на константу результатом выражения, тип которого может быть преобразован (см. раздел 2.1.2) в тип ссылки. В частности, мы можем связать ссылку на константу с неконстантным объектом, литералом или более общим выражением:

```
int i = 42;
const int &r1 = i;           // можно связать ссылку
const int& с обычным
                           // объектом int
```

```
const int &r2 = 42; // ok: r1 - ссылка на константу
const int &r3 = r1 * 2; // ok: r3 - ссылка на константу
int &r4 = r * 2; // ошибка: r4 - простая, неконстантная ссылка
```

Простейший способ понять это различие в правилах инициализации — рассмотреть то, что происходит при связывании ссылки с объектом другого типа:

```
double dval = 3.14;
const int &ri = dval;
```

Здесь ссылка *ri* ссылается на переменную типа *int*. Операции со ссылкой *ri* будут целочисленными, но переменная *dval* содержит число с плавающей запятой, а не целое число. Чтобы удостовериться в том, что объект, с которым связана ссылка *ri*, имеет тип *int*, компилятор преобразует этот код в нечто следующее:

```
const int temp = dval; // создать временную константу типа int из // переменной типа double
const int &ri = temp; // связать ссылку ri с временной константой
```

В данном случае ссылка *ri* связана с временным объектом (*temporary*). Временный объект — это безымянный объект, создаваемый компилятором для хранения промежуточного результата вычисления. Программисты C++ зачастую используют слово "temporary" как сокращение термина "temporary object".

Теперь рассмотрим, что могло бы произойти, будь инициализация позволена, но ссылка *ri* не была бы константной. В этом случае мы могли бы присвоить значение по ссылке *ri*. Это изменило бы объект, с которым связана ссылка *ri*. Этот временный объект имеет тип не *dval*. Программист, заставивший ссылку *ri* ссылаться на переменную *dval*, вероятно, ожидал, что присвоение по ссылке *ri* изменит переменную *dval*. В конце концов, почему произошло присвоение по ссылке *ri*, если не было намерения изменять объект, с которым она связана? Поскольку связь ссылки с временным объектом осуществляется уж конечно не то, что подразумевал программист, язык считает это некорректным.

Ссылка на константу может ссылаться на неконстантный объект

Важно понимать, что ссылка на константу ограничивает только то, что при помощи этой ссылки можно делать. Привязка ссылки к константному объекту ничего не говорит о том, является ли сам основной объект константой. Поскольку основной объект может оказаться неконстантным, он может быть изменен другими способами:

```
int i = 42;
int &r1 = i;           // r1 связана с i
const int &r2 = i;    // r2 тоже связана с i;
                      // но она не может
использоваться для изменения i
r1 = 0;               // r1 - неконстантна; i теперь 0
r2 = 0;               // ошибка: r2 - ссылка на
константу
```

Привязка ссылки `r2` к неконстантной переменной `i` типа `int` вполне допустима. Но ссылку `r2` нельзя использовать для изменения значения переменной `i`. Несмотря на это, значение переменной `i` вполне можно изменить другим способом. Например, можно присвоить ей значение непосредственно или при помощи другой связанной с ней ссылки, такой как `r1`.



2.4.2. Указатели и спецификатор `const`

Подобно ссылкам, вполне возможно определять указатели, которые указывают на объект константного или неконстантного типа. Как и ссылку на константу (см. раздел 2.4.1), указатель на константу (pointer to `const`) невозможно использовать для изменения объекта, на который он указывает. Адрес константного объекта можно хранить только в указателе на константу:

```
const double pi = 3.14;      // pi - константа; ее
значение неизменно
double *ptr = &pi;          // ошибка: ptr - простой
указатель
const double *cptr = &pi;    // ok: cptr может
указывать на тип
                           // const double
*cptr = 42;                // ошибка: нельзя
```

*присвоить `*cptr`*

В разделе 2.3.2 упоминалось о наличии двух исключений из правила, согласно которому типы указателя и объекта, на который он указывает, должны совпадать. Первое исключение — это возможность использования указателя на константу для указания на неконстантный объект:

```
double dval = 3.14; // dval типа double; ее  
значение неизменно  
cptr = &dval; // ok: но изменить dval при  
помощи cptr нельзя
```

Подобно ссылке на константу, указатель на константу ничего не говорит о том, является ли объект, на который он указывает, константой. Определение указателя как указателя на константу влияет только на то, что с его помощью можно сделать. Не забывайте, что нет никакой гарантии того, что объект, на который указывает указатель на константу, не будет изменяться.



Возможно, указатели и ссылки на константы следует рассматривать как указатели или ссылки, "которые *полагают*, что они указывают или ссылаются на константы".

Константные указатели

В отличие от ссылок, указатели — это объекты. Следовательно, подобно любым другим объектам, вполне может быть указатель, сам являющийся константой. Как и любой другой константный объект, **константный указатель** следует инициализировать, после чего изменить его значение (т.е. адрес, который он содержит) больше нельзя. Константный указатель объявляют, расположив ключевое слово `const` после символа `*`. Это означает, что данный указатель является константой, а не обычным указателем на константу.

```
int errNumb = 0;  
int *const curErr = &errNumb; // curErr всегда  
будет указывать на errNumb  
const double pi = 3.14159;  
const double *const pip = &pi; // pip константный  
указатель на  
// константный
```

объект

Как уже упоминалось в разделе 2.3.3, проще всего понять эти объявления, читая их справа налево. В данном случае ближе всего к имени `curErr` расположен спецификатор `const`, означая, что сам объект `curErr` будет константным. Тип этого объекта формирует остальная часть оператора объявления. Следующий символ оператора объявления, `*`, означает, что `curErr` — это константный указатель. И наконец, объявление завершает базовый тип, означая, что `curErr` — это константный указатель на объект типа `int`. Аналогично `rip` — это константный указатель на объект типа `const double`.

Тот факт, что указатель сам является константой, ничто не говорит о том, можем ли мы использовать указатель для изменения основного объекта. Возможность изменения объекта полностью зависит от типа, на который указывает указатель. Например, `rip` — это константный указатель на константу. Ни значение объекта, на который указывает указатель `rip`, ни хранящийся в нем адрес не могут быть изменены. С другой стороны, указатель `curErr` имеет простой, неконстантный тип `int`. Указатель `curErr` можно использовать для изменения значения переменной `errNumb`:

```
*rip = 2.72; // ошибка: rip - указатель на
константу
// если значение объекта, на который указывает
указатель curErr
// (т. е. errNumb), отлично от нуля
if (*curErr) {
    errorHandler();
    *curErr = 0; // обнулить значение объекта, на
который
                // указывает указатель curErr
}
```

Упражнения раздела 2.4.2

Упражнение 2.27. Какие из следующих инициализаций допустимы? Объясните почему.

- (a) `int i = -1, &r = 0;`
- (b) `int *const p2 = &i2;`
- (c) `const int i = -1, &r = 0;`
- (d) `const int *const p3 = &i2;`

(e) const int *p1 = &i2; (f) const int &const
r2;
(g) const int i2 = i, &r = i;

Упражнение 2.28. Объясните следующие определения. Какие из них недопустимы?

- (a) int i, *const cp; (b) int *p1, *const p2;
(c) const int ic, &r = ic; (d) const int *const p3;
(e) const int *p;

Упражнение 2.29. С учетом переменных из предыдущих упражнений, какие из следующих присвоений допустимы? Объясните почему.

- (a) i = ic; (b) pi = p3;
(c) pi = ⁣ (d) p3 = ⁣
(e) p2 = pi; (f) ic = *p3;



2.4.3. Спецификатор `const` верхнего уровня

Как уже упоминалось, указатель — это объект, способный указывать на другой объект. В результате можно сразу сказать, является ли указатель сам константой и являются ли константой объекты, на которые он может указывать. Термин *спецификатор const верхнего уровня* (top-level `const`) используется для обозначения того ключевого слова `const`, которое объявляет константой сам указатель. Когда указатель способен указывать на константный объект, это называется *спецификатор const нижнего уровня* (low-level `const`).

В более общем смысле спецификатор `const` верхнего уровня означает, что объект сам константа. Спецификатор `const` верхнего уровня может присутствовать в любом типе объекта, будь то один из встроенных арифметических типов, тип класса или ссылочный тип. Спецификатор `const` нижнего уровня присутствует в базовом типе составных типов, таких как указатели или ссылки. Обратите внимание, что ссылочные типы, в отличие от большинства других типов, способны иметь спецификаторы `const` как верхнего, так и нижнего уровня, независимо друг от друга.

```
int i = 0;
int *const pi = &i; // нельзя изменить значение
pi; // const верхнего уровня
```

```
const int ci = 42;      // нельзя изменить ci; const
верхнего уровня
const int *p2 = &ci;   // нельзя изменить p2; const
нижнего уровня
const int *const p3 = p2; // справа const верхнего
уровня, слева нет
const int &r = ci;     // const в ссылочных типах
всегда нижнего уровня
```



Различие между спецификаторами `const` верхнего и нижнего уровней проявляется при копировании объекта. При копировании объекта спецификатор `const` верхнего уровня игнорируется.

```
i = ci;      // ok: копирование значения ci;
спецификатор const верхнего
// уровня в ci игнорируется
p2 = p3;     // ok: указываемые типы совпадают;
спецификатор const верхнего
// уровня в p3 игнорируется
```

Копирование объекта не изменяет копируемый объект. Поэтому независимо, является ли копируемый или копирующий объект константой.

Спецификатор `const` нижнего уровня, напротив, никогда не игнорируется. При копировании объектов у них обоих должны быть одинаковые спецификаторы `const` нижнего уровня, или должно быть возможно преобразование между типами этих двух объектов. Как правило, преобразование неконстанты в константу возможно, но не наоборот.

```
int *p = p3;           // ошибка: p3 имеет const
нижнего уровня, а p - нет
p2 = p3;               // ok: p2 имеет то же const
нижнего уровня, что и p3
p2 = &i;                // ok: преобразование int* в
const int* возможно
int &r = ci;            // ошибка: невозможно связать
обычную int& с
// объектом const int
const int &r2 = i; // ok: const int& можно связать
с обычным int
```

У указателя `p3` есть спецификатор `const` нижнего и верхнего уровня. При копировании указателя `p3` можно проигнорировать его спецификатор `const` верхнего уровня, но не тот факт, что он указывает на константный тип. Следовательно, нельзя использовать указатель `p3` для инициализации указателя `p`, который указывает на простой (неконстантный) тип `int`. С другой стороны, вполне можно присвоить указатель `p3` указателю `p2`. У обоих указателей тот же тип (спецификатор `const` нижнего уровня). Тот факт, что `p3` — константный указатель (т.е. у него есть спецификатор `const` верхнего уровня), не имеет значения.

Упражнения раздела 2.4.3

Упражнение 2.30. Укажите по каждому из следующих объявлений, имеет ли объявляемый объект спецификатор `const` нижнего или верхнего уровня.

```
const int v2 = 0;
int v1 = v2;
int *p1 = &v1, &r1 = v1;
const int *p2 = &v2, *const p3 = &i, &r2 = v2;
```

Упражнение 2.31. С учетом объявлений в предыдущем упражнении укажите, допустимы ли следующие присвоения. Объясните, как спецификатор `const` верхнего или нижнего уровня применяется в каждом случае.

```
r1 = v2;
p1 = p2; p2 = p1;
p1 = p3; p2 = p3;
```



2.4.4. Переменные `constexpr` и константные выражения

Константное выражение (`constant expression`) — это выражение, значение которого не может измениться и вычисляется во время компиляции. Литерал — это константное выражение. Константный объект, инициализируемый константным выражением, также является константным выражением. Вскоре мы увидим, что в языке есть несколько контекстов, требующих константных выражений.

Является ли данный объект (или выражение) константным

выражением, зависит от типов и инициализаторов. Например:

```
const int max_files = 20; // max_files -  
константное выражение  
const int limit = max_files + 1; // limit -  
константное выражение  
int staff_size = 27; // staff_size -  
неконстантное выражение  
const int sz = get_size(); // sz - неконстантное  
выражение
```

Хотя переменная `staff_size` инициализируется литералом, это неконстантное выражение, поскольку он имеет обычный тип `int`, а не `const int`. С другой стороны, хоть переменная `sz` и константа, значение ее инициализатора неизвестно до времени выполнения. Следовательно, это неконстантное выражение.

Переменные constexpr

В большой системе может быть трудно утверждать (наверняка), что инициализатор — константное выражение. Константная переменная могла бы быть определена с инициализатором, который мы полагаем константным выражением. Однако при использовании этой переменной в контексте, требующем константного выражения, может оказаться, что инициализатор не был константным выражением. Как правило, определение объекта и его использования в таком контексте располагаются довольно далеко друг от друга.



Согласно новому стандарту, можно попросить компилятор проверить, является ли переменная константным выражением, использовав в ее объявлении ключевое слово `constexpr`. *Переменные constexpr* неявно являются константой и должны инициализироваться константными выражениями.

```
constexpr int mf = 20; // 20 - константное  
выражение  
constexpr int limit = mf + 1; // mf + 1 -  
константное выражение  
constexpr int sz = size(); // допустимо, только  
если size() является  
// функцией constexpr
```

Хоть и нельзя использовать обычную функцию как инициализатор для переменной `constexpr`, как будет описано в разделе 6.5.2, новый стандарт позволяет определять функции как `constexpr`. Такие функции должны быть достаточно просты, чтобы компилятор мог выполнить их во время компиляции. Функции `constexpr` можно использовать в инициализаторе переменной `constexpr`.



Рекомендуем

Как правило, ключевое слово `constexpr` имеет смысл использовать для переменных, которые предполагается использовать как константные выражения.

Литеральные типы

Поскольку константное выражение обрабатывается во время компиляции, есть пределы для типов, которые можно использовать в объявлении `constexpr`. Типы, которые можно использовать в объявлении `constexpr`, известны как *литеральные типы* (literal type), поскольку они достаточно просты для литературальных значений.

Все использованные до сих пор типы — арифметический, ссылка и указатель — это литературные типы. Наш класс `Sales_item` и библиотечный тип `string` не относятся к литературным типам. Следовательно, нельзя определить переменные этих типов как `constexpr`. Другие виды литературных типов рассматриваются в разделах 7.5.6 и 19.3.

Хотя указатели и ссылки можно определить как `constexpr`, используемые для их инициализации объекты жестко ограничены. Указатель `constexpr` можно инициализировать литералом `nullptr` или литералом (т.е. константным выражением) 0. Можно также указать на (или связать с) объект, который остается по фиксированному адресу.

По причинам, рассматриваемым в разделе 6.1.1, определенные в функции переменные обычно не хранятся по фиксированному адресу. Следовательно, нельзя использовать указатель `constexpr` для указания на такие переменные. С другой стороны, адрес объекта, определенного вне любой функции, является константным выражением и, таким образом, может использоваться для инициализации указателя `constexpr`. Как будет описано в разделе 6.1.1, функции могут определять переменные,

существующие на протяжении нескольких вызовов этой функции. Как и объект, определенный вне любой функции, эти специальные локальные объекты также имеют фиксированные адреса. Поэтому и ссылка `constexpr` может быть связана с такой переменной, и указатель `constexpr` может содержать ее адрес.

Указатели и спецификатор `constexpr`

Важно понимать, что при определении указателя в объявлении `constexpr` спецификатор `constexpr` относится к указателю, а не к типу, на который указывает указатель.

```
const int *p = nullptr;           // p - указатель на  
const int  
constexpr int *q = nullptr;      // q - константный  
указатель на int
```

Несмотря на внешний вид, типы `p` и `q` весьма различны; `p` — указатель на константу, тогда как `q` — константный указатель. Различие является следствием того факта, что спецификатор `constexpr` налагает на определяемый объект спецификатор `const` верхнего уровня (см. раздел 2.4.3).

Как и любой другой константный указатель, указатель `constexpr` может указать на константный или неконстантный тип.

```
constexpr int *np = nullptr;     // np - нулевой  
константный указатель  
                                // на int  
  
int j = 0;  
constexpr int i = 42;            // типом i является  
const int  
// i и j должны быть определены вне любой функции  
constexpr const int *p = &i;    // p - константный  
указатель  
                                // на const int i  
constexpr int *p1 = &j;         // p1 - константный  
указатель на int j
```

Упражнения раздела 2.4.4

Упражнение 2.32. Допустим ли следующий код? Если нет, то как его исправить?

```
int null = 0, *p = null;
```

2.5. Работа с типами

По мере усложнения программ используемые в них типы также становятся все более сложными. Осложнения в использовании типов возникают по двум причинам. Имена некоторых типов трудно писать по памяти. Написание некоторых их форм утомительно и подвержено ошибкам. Кроме того, формат записи сложного типа способен скрыть его цель или значение. Другой источник осложнений кроется в том, что иногда трудно точно определить необходимый тип. Это может потребовать оглянуться на контекст программы.

2.5.1. Псевдонимы типов

Псевдоним типа (type alias) — это имя, являющееся синонимом имени другого типа. Псевдонимы типа позволяют упростить сложные определения типов, облегчая их использование. Псевдонимы типа позволяют также подчеркивать цель использования типа. Определить псевдоним типа можно одним из двух способов. Традиционно он определяется при помощи ключевого слова `typedef`:

```
typedef double wages;      // wages - синоним для  
double  
typedef wages base, *p;    // base - синоним для  
double, а p - для double*
```

Ключевое слово `typedef` может быть частью базового типа в объявлении (см. раздел 2.3). Объявления, включающие ключевое слово `typedef`, определяют псевдонимы типа, а не переменные. Как и в любое другое объявление, в это можно включать модификаторы типа, которые определяют составные типы, включающие базовый тип.



Новый стандарт вводит второй способ определения псевдонима типа при помощи *объявления псевдонима* (alias declaration) и знака `=`.

```
using SI = Sales_item; // SI - синоним для  
Sales_item
```

Объявление псевдонима задает слева от оператора `=` имя псевдонима типа, который расположен справа.

Псевдоним типа — это имя типа, оно может присутствовать везде, где присутствует имя типа.

```
wages hourly, weekly; // то же, что и double  
hourly, weekly;  
SI item; // то же, что и Sales_item  
item
```



Указатели, константы и псевдонимы типа

Объявления, использующие псевдонимы типа, представляющие составные типы и константы, могут приводить к удивительным результатам. Например, следующие объявления используют тип `pstring`, который является псевдонимом для типа `char*`.

```
typedef char *pstring;  
const pstring cstr = 0; // cstr - константный  
указатель на char  
const pstring *ps; // ps - указатель на  
константный указатель  
// на тип char
```

Базовым типом в этих объявлениях является `const pstring`. Как обычно, модификатор `const` в базовом типе модифицирует данный тип. Тип `pstring` — это указатель на тип `char`, а `const pstring` — это константный указатель на тип `char`, но не указатель на тип `const char`.

Заманчиво, хоть и неправильно, интерпретировать объявление, которое использует псевдоним типа как концептуальную замену псевдонима, соответствующим ему типом:

```
const char *cstr = 0; // неправильная интерпретация  
const pstring cstr
```

Однако эта интерпретация неправильна. Когда используется тип `pstring` в объявлении, базовым типом объявления является тип указателя. При перезаписи объявления с использованием `char*`, базовым типом будет `char`, а `*` будет частью оператора объявления. В данном случае базовый тип — это `const char`. Перезапись объявляет `cstr` указателем на тип `const char`, а не константным указателем на тип `char`.



2.5.2. Спецификатор типа `auto`



Нет ничего необычного в желании сохранить значение выражения в переменной. Чтобы объявить переменную, нужно знать тип этого выражения. Когда мы пишем программу, может быть на удивление трудно (а иногда даже невозможно) определить тип выражения. По новому стандарту можно позволить компилятору самому выяснить этот тип. Для этого используется *спецификатор типа auto*. В отличие от таких спецификаторов типа, как `double`, задающих определенный тип, спецификатор `auto` приказывает компилятору вывести тип из инициализатора. Само собой разумеется, у переменной, использующей спецификатор типа `auto`, должен быть инициализатор.

```
// тип item выводится из типа результата суммы val1
и val2
auto item = val1 + val2; // item инициализируется
результатом val1 + val2
```

Здесь компилятор выведет тип переменной `item` из типа значения, возвращенного при применении оператора `+` к переменным `val1` и `val2`. Если переменные `val1` и `val2` — объекты класса `Sales_item` (см. раздел 1.5), типом переменной `item` будет класс `Sales_item`. Если эти переменные имеют тип `double`, то у переменной `item` будет тип `double` и т.д.

Подобно любому другому спецификатору типа, используя спецификатор `auto`, можно определить несколько переменных. Поскольку объявление может задействовать только один базовый тип, у инициализаторов всех переменных в объявлении должны быть типы, совместимые друг с другом.

```
auto i = 0, *p = &i;           // ok: i - int, а p -
указатель на int
auto sz = 0, pi = 3.14; // ошибка: несовместимые
типы у sz и pi
```

Составные типы, const и auto

Выводимый компилятором тип для спецификатора `auto` не всегда точно совпадает с типом инициализатора. Компилятор корректирует тип так, чтобы он соответствовал обычным правилам инициализации.

Во-первых, как уже упоминалось, при использовании ссылки в действительности используется объект, на который она ссылается. В частности, при использовании ссылки как инициализатора им является соответствующий объект. Компилятор использует тип этого объекта для выводения типа `auto`.

```
int i = 0, &r = i;
auto a = r; // a - int (r - псевдоним для i,
имеющий тип int)
```

Во-вторых, выводение типа `auto` обычно игнорирует спецификаторы `const` верхнего уровня (см. раздел 2.4.3). Как обычно в инициализациях, спецификаторы `const` нижнего уровня учитываются в случае, когда инициализатор является указателем на константу.

```
const int ci = i, &cr = ci;
auto b = ci; // b - int (const верхнего уровня в
ci отброшен)
auto c = cr; // c - int (cr - псевдоним для ci с
const верхнего
                // уровня)
auto d = &i; // d - int* (& объекта int - int*)
auto e = &ci; // e - const int* (& константного
объекта - const нижнего
                // уровня)
```

Если необходимо, чтобы у выведенного типа был спецификатор `const` верхнего уровня, его следует указать явно.

```
const auto f = ci; // выведенный тип ci - int; тип
f - const int
```

Можно также указать, что необходима ссылка на автоматически выведенный тип. Обычные правила инициализации все еще применимы.

```
auto &g = ci; // g - const int&, связанный с ci
auto &h = 42; // ошибка: нельзя связать простую
ссылку с литералом
const auto &j = 42; // ok: константную ссылку с
литералом связать можно
```

Когда запрашивается ссылка на автоматически выведенный тип, спецификаторы `const` верхнего уровня в инициализаторе не игнорируются. Как обычно при связывании ссылки с инициализатором, спецификаторы `const` не относятся к верхнему уровню.

При определении нескольких переменных в том же операторе важно не

забывать, что ссылка или указатель — это часть специфического оператора объявления, а не часть базового типа объявления. Как обычно, инициализаторы должны быть совместимы с автоматически выведенными типами:

```
auto k = ci, &l = i;      // k - int; l - int&
auto &m = ci, *p = &ci; // m - const int&; p -
указатель на const int
// ошибка: выведение типа из i - int;
// тип, выведенный из &ci - const int
auto &n = i, *p2 = &ci;
```

Упражнения раздела 2.5.2

Упражнение 2.33. С учетом определения переменных из этого раздела укажите то, что происходит в каждом из этих присвоений.

```
a = 42; b = 42; c = 42;
d = 42; e = 42; g = 42;
```

Упражнение 2.34. Напишите программу, содержащую переменные и присвоения из предыдущего упражнения. Выведите значения переменных до и после присвоений, чтобы проверить правильность предположений в предыдущем упражнении. Если они неправильны, изучите примеры еще раз и выясните, что привело к неправильному заключению.

Упражнение 2.35. Укажите типы, выведенные в каждом из следующих определений. Затем напишите программу, чтобы убедиться в своей правоте.

```
const int i = 42;
auto j = i; const auto &k = i; auto *p = &i;
const auto j2 = i, &k2 = i;
```



2.5.3. Спецификатор типа `decltype`



Иногда необходимо определить переменную, тип которой компилятор выводит из выражения, но не использовать это выражение для инициализации переменной. Для таких случаев новый стандарт вводит спецификатор типа `decltype`, возвращающий тип его операнда. Компилятор анализирует выражение и определяет его тип, но не вычисляет

его результат.

```
decltype(f()) sum = x; // sum имеет тот тип,  
// который возвращает  
функция f
```

Здесь компилятор не вызывает функцию `f()`, но он использует тип, который вернёт такой вызов для переменной `sum`. Таким образом, компилятор назначает переменной `sum` тот же тип, который был бы возвращен при вызове функции `f()`.

Таким образом, спецификатор `decltype` учитывает спецификатор `const` верхнего уровня и ссылки, но несколько отличается от того, как работает спецификатор `auto`. Когда выражение, к которому применен спецификатор `decltype`, является переменной, он возвращает тип этой переменной, включая спецификатор `const` верхнего уровня и ссылки.

```
const int ci = 0, &cj = ci;  
decltype(ci) x = 0; // x имеет тип const int  
decltype(cj) y = x; // y имеет тип const int& и  
связана с x  
decltype(cj) z; // ошибка: z - ссылка, она должна  
быть инициализирована
```

Поскольку `cj` — ссылка, `decltype(cj)` — ссылочный тип. Как и любую другую ссылку, ссылку `z` следует инициализировать.

Следует заметить, что спецификатор `decltype` — единственный контекст, в котором переменная определена, поскольку ссылка не рассматривается как синоним объекта, на который она ссылается.



Спецификатор `decltype` и ссылки

Когда спецификатор `decltype` применяется к выражению, которое не является переменной, получаемый тип соответствует типу выражения. Как будет продемонстрировано в разделе 4.1.1, некоторые выражения заставят спецификатор `decltype` возвращать ссылочный тип. По правде говоря, спецификатор `decltype` возвращает ссылочный тип для выражений, результатом которых являются объекты, способные стоять слева от оператора присвоения.

```
// decltype выражение может быть ссылочным типом  
int i = 42, *p = &i, &r = i;
```

```

 decltype( r + 0 ) b; // ok: сложение возвращает тип
 int; b имеет тип int
                                // (не инициализирована)
 decltype( *p ) c;           // ошибка: c имеет тип int& и
 требует инициализации

```

Здесь `r` — ссылка, поэтому `decltype(r)` возвращает ссылочный тип. Если необходим тип, на который ссылается ссылка `r`, можно использовать ее в таком выражении, как `r + 0`, поскольку оно возвращает значение не ссылочного типа.

С другой стороны, оператор обращения к значению — пример выражения, для которого спецификатор `decltype` возвращает ссылку. Как уже упоминалось, при обращении к значению указателя возвращается объект, на который он указывает. Кроме того, этому объекту можно присвоить значение. Таким образом, `decltype(*p)` выведет тип `int&`, а не просто `int`.



Еще одно важное различие между спецификаторами `decltype` и `auto` в том, что выведение, осуществляемое спецификатором `decltype`, зависит от формы данного выражения. Не всегда понимают то, что включение имени переменной в круглые скобки влияет на тип, возвращаемый спецификатором `decltype`. При применении спецификатора `decltype` к переменной без круглых скобок получается тип этой переменной. Если заключить имя переменной в одни или несколько круглых скобок, то компилятор будет рассматривать operand как выражение. Переменная — это выражение, которое способно быть левым operandом присвоения. В результате спецификатор `decltype` для такого выражения возвратит ссылку.

```

 // decltype переменной в скобках - всегда ссылка
 decltype( ( i ) ) d; // ошибка: d - int& и должна
 инициализироваться
 decltype( i ) e;      // ok: e имеет тип int (не
 инициализирована)

```



Помните, что спецификатор `decltype((переменная))` (обратите

внимание на парные круглые скобки) *всегда* возвращает ссылочный тип, а спецификатор `decltype(переменная)` возвращает ссылочный тип, только если *переменная* является ссылкой.

Упражнения раздела 2.5.3

Упражнение 2.36. Определите в следующем коде тип каждой переменной и значения, которые будет иметь каждая из них по завершении.

```
int a = 3, b = 4;
decltype(a) c = a;
decltype((b)) d = a;
++c;
++d;
```

Упражнение 2.37. Присвоение — это пример выражения, которое возвращает ссылочный тип. Тип — это ссылка на тип левого операнда. Таким образом, если переменная *i* имеет тип `int`, то выражение *i* = *x* имеет тип `int&`. С учетом этого определите тип и значение каждой переменной в следующем коде:

```
int a = 3, b = 4;
decltype(a) c = a;
decltype(a = b) d = a;
```

Упражнение 2.38. Опишите различия выводения типа спецификаторами `decltype` и `auto`. Приведите пример выражения, где спецификаторы `auto` и `decltype` выведут тот же тип, и пример, где они выведут разные типы.



2.6. Определение собственных структур данных

На самом простом уровне *структура данных* (data structure) — это способ группировки взаимосвязанных данных и стратегии их использования. Например, класс `Sales_item` группирует ISBN книги, количество проданных экземпляров и выручку от этой продажи. Он предоставляет также набор операций, таких как функция `isbn()` и операторы `>>`, `<<`, `+` и `+=`.

В языке C++ мы создаем собственные типы данных, определяя класс. Такие библиотечные типы, как `string`, `istream` и `ostream`, определены как классы, подобно типу `Sales_item` в главе 1. Поддержка классов в языке C++ весьма обширна, фактически части III и IV в значительной степени посвящены описанию средств, связанных с классами. Хотя класс `Sales_item` довольно прост, мы не сможем определить его полностью, пока не узнаем в главе 14, как писать собственные операторы.



2.6.1 Определение типа `Sales_data`

Несмотря на то что мы еще не можем написать свой класс `Sales_item` полностью, уже вполне можно создать достаточно реалистичный класс, группирующий необходимые элементы данных. Стратегия использования этого класса заключается в том, что пользователи будут получать доступ непосредственно к элементам данных и смогут самостоятельно реализовать необходимые операции.

Поскольку создаваемая структура данных не поддерживает операций, назовем новую версию `Sales_data`, чтобы отличать ее от типа `Sales_item`. Определим класс следующим образом:

```
struct Sales_data {  
    std::string bookNo;  
    unsigned units_sold = 0;  
    double revenue = 0.0;  
};
```

Класс начинается с ключевого слова `struct`, сопровождаемого именем класса и (возможно пустым) телом класса. Тело класса заключено в фигурные скобки и формирует новую область видимости (см. раздел 2.2.4). Определенные в классе имена должны быть уникальны в пределах класса, но вне класса они могут повторяться.

Ближайшие фигурные скобки, заключающие тело класса, следует сопроводить точкой с запятой. Точка с запятой необходима, так как после тела класса можно определить переменные:

```
struct Sales_data { /* ... */ } accum, trans,  
*salesptr;  
// эквивалентно, но лучше определять эти объекты  
так  
struct Sales_data { /* ... */ };  
Sales data accum, trans, *salesptr;
```

Точка с запятой отмечает конец (обычно пустого) списка объявления операторов. Обычно определение объекта в составе определения класса — это не лучшая идея. Объединение в одном операторе определений двух разных сущностей (класса и переменной) ухудшает читабельность кода.



Забытая точка с запятой в конце определения класса — довольно распространенная ошибка начинающих программистов.

Переменные-члены класса

В теле класса определены *члены* (member) класса. У нашего класса есть только *переменные-члены* (data member). Переменные-члены класса определяют содержимое объектов этого класса. Каждый объект обладает собственным экземпляром переменных-членов класса. Изменение переменных-членов одного объекта не изменяет данные в любом другом объекте класса `Sales_data`.

Переменные-члены определяются точно так же, как и обычные переменные: указывается базовый тип, затем список из одного или нескольких операторов объявления. У нашего класса будут три переменные-члены: член типа `string` по имени `bookNo`, член типа `unsigned` по имени `units_sold` и член типа `double` по имени `revenue`. Эти три переменные-члены будут у каждого объекта класса `Sales_data`.

По новому стандарту переменной-члену можно предоставить *внутриклассовый инициализатор* (in-class initializer). Он используется для инициализации переменных-членов при создании объектов. Члены без инициализатора инициализируются по умолчанию (см. раздел 2.2.1). Таким образом, при определении объектов класса `Sales_data` переменные-члены `units_sold` и `revenue` будут инициализированы значением 0, а переменная-член `bookNo` — пустой строкой.

Внутриклассовые инициализаторы ограничены формой их использования (см. раздел 2.2.1): они должны либо быть заключены в фигурные скобки, либо следовать за знаком `=`. Нельзя определить внутриклассовый инициализатор в круглых скобках.

В разделе 7.2 указано, что язык C++ обладает еще одним ключевым словом, `class`, также используемым для определения собственной структуры данных. В этом разделе используем ключевое слово `struct`, поскольку пока еще не рассмотрены приведенные в главе 7 дополнительные средства, связанные с классом.

Упражнения раздела 2.6.1

Упражнение 2.39. Откомпилируйте следующую программу и посмотрите, что будет, если не поставить точку с запятой после определения класса. Запомните полученное сообщение, чтобы узнать его в будущем.

```
struct Foo { /* пусто */ } // Примечание: нет точки
с запятой
```

```
int main() {
    return 0;
}
```

Упражнение 2.40. Напишите собственную версию класса `Sales_data`.



2.6.2. Использование класса `Sales_data`

В отличие от класса `Sales_item`, класс `Sales_data` не поддерживает операций. Пользователи класса `Sales_data` должны сами

писать все операции, в которых они нуждаются. В качестве примера напишем новую версию программы из раздела 1.5.2, которая выводила сумму двух транзакций. Программа будет получать на входе такие транзакции:

```
0-201-78345-X 3 20.00  
0-201-78345-X 2 25.00
```

Каждая транзакция содержит ISBN, количество проданных книг и цену, по которой была продана каждая книга.

Суммирование двух объектов класса Sales_data

Поскольку класс Sales_data не предоставляет операций, придется написать собственный код, осуществляющий ввод, вывод и сложение. Будем подразумевать, что класс Sales_data определен в заголовке Sales_data.h. Определение заголовка рассмотрим в разделе 2.6.3.

Так как эта программа будет длиннее любой, написанной до сих пор, рассмотрим ее по частям. В целом у программы будет следующая структура:

```
#include <iostream>  
#include <string>  
#include "Sales_data.h"  
int main() {  
    Sales_data data1, data2;  
    // код чтения данных в data1 и data2  
    // код проверки наличия у data1 и data2  
одинакового ISBN  
    // если это так, то вывести сумму data1 и data2  
}
```

Как и первоначальная программа, эта начинается с включения заголовков, необходимых для определения переменных, содержащих ввод. Обратите внимание, что, в отличие от версии Sales_item, новая программа включает заголовок string. Он необходим потому, что код должен манипулировать переменной-членом bookNo типа string.

Чтение данных в объект класса Sales_data

Хотя до глав 3 и 10 мы не будем описывать библиотечный тип string подробно, упомянем пока лишь то, что необходимо знать для определения и использования члена класса, содержащего ISBN. Тип string содержит последовательность символов. Он имеет операторы >>, << и == для

чтения, записи и сравнения строк соответственно. Этих знаний достаточно для написания кода чтения первой транзакции.

```
double price = 0; // цена за книгу, используемая
для вычисления
        // общей выручки
// читать первую транзакцию:
// ISBN, количество проданных книг, цена книги
std::cin >> data1.bookNo >> data1.units_sold >>
price;
// вычислить общий доход из price и units_sold
data1.revenue = data1.units_sold * price;
```

Транзакции содержат цену, по которой была продана каждая книга, но структура данных хранит общий доход. Данные транзакции будем читать в переменную `price` (цена) типа `double`, исходя из которой и вычислим член `revenue` (доход).

```
std::cin >> data1.bookNo >> data1.units_sold >>
price;
```

Для чтения значений членов `bookNo` и `units_sold` (продано экземпляров) объекта по имени `data1` оператор ввода использует точечный оператор (см. раздел 1.5.2).

Последний оператор присваивает произведение `data1.units_sold` и `price` переменной-члену `revenue` объекта `data1`.

Затем программа повторяет тот же код для чтения данных в объект `data2`.

```
// читать вторую транзакцию
std::cin >> data2.bookNo >> data2.units_sold >>
price;
data2.revenue = data2.units_sold * price;
```

Вывод суммы двух объектов класса Sales_data

Следующая задача — проверить наличие у транзакций одинакового ISBN. Если это так, вывести их сумму, в противном случае отобразить сообщение об ошибке.

```
if (data1.bookNo == data2.bookNo) {
    unsigned totalCnt = data1.units_sold +
data2.units_sold;
    double totalRevenue = data1.revenue +
data2.revenue;
```

```

    // вывести ISBN, общее количество проданных
    // экземпляров,
    // общий доход, среднюю цену за книгу
    std::cout << data1.bookNo << " " << totalCnt
        << " " << totalRevenue << " ";
    if (totalCnt != 0)
        std::cout << totalRevenue/totalCnt << std::endl;
    else
        std::cout << "(no sales)" << std::endl;
    return 0; // означает успех
} else { // транзакции не для того же ISBN
    std::cerr << "Data must refer to the same ISBN"
        << std::endl;
    return -1; // означает неудачу
}

```

Первый оператор `if` сравнивает члены `bookNo` объектов `data1` и `data2`. Если эти члены содержат одинаковый ISBN, выполняется код в фигурных скобках, суммирующий компоненты двух переменных. Поскольку необходимо вывести среднюю цену, сначала вычислим общее количество проданных экземпляров и общий доход, а затем сохраним их в переменных `totalCnt` и `totalRevenue` соответственно. Выводим эти значения, а затем проверяем, были ли книги проданы, и если да, то выводим вычисленную среднюю цену за книгу. Если никаких продаж не было, выводим сообщение, обращающее внимание на этот факт.

Упражнения раздела 2.6.2

Упражнение 2.41. Используйте класс `Sales_data` для перезаписи кода упражнений из разделов 1.5.1, 1.5.2 и 1.6. А также определите свой класс `Sales_data` в том же файле, что и функция `main()`.



2.6.3. Создание собственных файлов заголовка

Как будет продемонстрировано в разделе 19.7, класс можно определить в функции, однако такие классы ограничены по функциональным возможностям. Поэтому классы обычно не определяют в функциях. При определении класса за пределами функции в каждом файле исходного кода

может быть только одно определение этого класса. Кроме того, если класс используется в нескольких разных файлах, определение класса в каждом файле должно быть тем же.

Чтобы гарантировать совпадение определений класса в каждом файле, классы обычно определяют в файлах заголовка. Как правило, классы хранятся в заголовках, имя которых совпадает с именем класса. Например, библиотечный тип `string` определен в заголовке `string`. Точно так же, как уже было продемонстрировано, наш класс `Sales_data` определен в файле заголовка `Sales_data.h`.

Заголовки (обычно) содержат сущности (такие как определения класса или переменных `const` и `constexpr` (см. раздел 2.4), которые могут быть определены в любом файле только однажды. Однако заголовки нередко должны использовать средства из других заголовков. Например, поскольку у класса `Sales_data` есть член типа `string`, заголовок `Sales_data.h` должен включать заголовок `string`. Как уже упоминалось, программы, использующие класс `Sales_data`, должны также включать заголовок `string`, чтобы использовать член `bookNo`. В результате использующие класс `Sales_data` программы будут включать заголовок `string` дважды: один раз непосредственно и один раз как следствие включения заголовка `Sales_data.h`. Поскольку заголовок мог бы быть включен несколько раз, код необходимо писать так, чтобы обезопасить от многократного включения.



После внесения любых изменений в заголовок необходимо перекомпилировать все использующие его файлы исходного кода, чтобы вступили в силу новые или измененные объявления.

Краткое введение в препроцессор

Наиболее распространенный способ обезопасить заголовок от многократного включения подразумевает использование препроцессора. *Препроцессор* (preprocessor), унаследованный языком C++ от языка C, является программой, которая запускается перед компилятором и изменяет исходный текст программ. Наши программы уже полагаются на такое средство препроцессора, как директива `#include`. Когда препроцессор встречает директиву `#include`, он заменяет ее содержимым указанного

заголовка.

Программы C++ используют также препроцессор для *защиты заголовка* (header guard). Защита заголовка полагается на переменные препроцессора (см. раздел 2.3.2). Переменные препроцессора способны находиться в одном из двух состояний: она либо определена, либо не определена. Директива `#define` получает имя и определяет его как переменную препроцессора. Есть еще две директивы, способные проверить, определена ли данная переменная препроцессора или нет. Директива `#ifdef` истинна, если переменная была определена, а директива `#ifndef` истинна, если переменная *не была* определена. В случае истинности проверки выполняется все, что расположено после директивы `#ifdef` или `#ifndef` и до соответствующей директивы `#endif`.

Эти средства можно использовать для принятия мер против множественного включения следующим образом:

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
#include <string>
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
#endif
```

При первом включении заголовка `Sales_data.h` директива `#ifndef` истинна, и препроцессор обработает строки после нее до директивы `#endif`. В результате переменная препроцессора `SALES_DATA_H` будет определена, а содержимое заголовка `Sales_data.h` скопировано в программу. Если впоследствии включить заголовок `Sales_data.h` в тот же файл, то директива `#ifndef` окажется ложна и строки между ней и директивой `#endif` будут проигнорированы.



Имена переменных препроцессора не подчиняются правилам областей видимости языка C++.

Переменные препроцессора, включая имена для защиты заголовка, должны быть уникальными во всей программе. Обычно мы гарантируем

的独特性是通过在头文件中包含类名来实现的。为了避免与程序中的其他实体发生冲突，通常将变量名全部大写。

Рекомендуем

У заголовков должна быть защита, даже если они не включаются в другие заголовки. Защита заголовка проста в написании, и при привычном их определении не нужно размышлять, нужны они или нет.

Упражнения раздела 2.6.3

Упражнение 2.42. Напишите собственную версию заголовка `Sales_data.h` и используйте его для новой версии упражнения из раздела 2.6.2.

Резюме

Типы — фундаментальная часть всех программ C++.

Каждый тип определяет требования по хранению и операциям, которые можно выполнять с объектами этого типа. Язык предоставляет набор фундаментальных встроенных типов, таких как `int` и `char`, которые тесно связаны с их представлением на аппаратных средствах машины. Типы могут быть неконстантными или константными; константный объект следует инициализировать. Будучи однажды инициализированным, значение константного объекта не может быть изменено. Кроме того, можно определить составные типы, такие как указатели или ссылки. Составной тип — это тип, определенный в терминах другого типа.

Язык позволяет определять собственные типы, т.е. классы. Библиотека использует классы, чтобы предоставить набор таких высокоуровневых абстракций, как типы `IO` и `string`.

Термины

Адрес (address). Номер байта в памяти, начиная с которого располагается объект.

Арифметический тип (arithmetic type). Встроенные типы, представляющие логические значения, символы, целые числа и числа с плавающей запятой.

Базовый тип (base type). Спецификатор типа, возможно со спецификатором `const`, который предшествует оператору объявления в объявлении. Базовый тип представляет общий тип, на основании которого строятся операторы объявления в объявлении.

Байт (byte). Наименьший адресуемый блок памяти. На большинстве машин байт составляет 8 битов.

Беззнаковый тип (unsigned). Целочисленный тип данных, переменные которого способны хранить значения больше или равные нулю.

В области видимости (in scope). Имя, которое видимо от текущей области видимости.

Внешняя область видимости (outer scope). Область видимости, включающая другую область видимости.

Внутриклассовый инициализатор (in-class initializer). Инициализатор, предоставленный как часть объявления переменной-члена класса. За внутриклассовым инициализатором следует символ `=`, или он заключается в фигурные скобки.

Временный объект (temporary). Безымянный объект, создаваемый компилятором при вычислении выражения. Временный объект существует до конца вычисления всего выражения, для которого он был создан.

Глобальная область видимости (global scope). Область видимости, внешняя для всех остальных областей видимости.

Директива препроцессора `#define`. Определяет переменную препроцессора.

Директива препроцессора `#endif`. Завершает область `#ifdef` или `#ifndef`.

Директива препроцессора `#ifdef`. Выясняет, что данная переменная определена.

Директива препроцессора `#ifndef`. Выясняет, что данная переменная не определена.

Защита заголовка (header guard). Переменная препроцессора, предназначенная для предотвращения неоднократного подключения содержимого заголовка в один файл исходного кода.

Знаковый тип (signed). Целочисленный тип данных, переменные которого способны хранить отрицательные и положительные числа, включая нуль.

Идентификатор (identifier). Последовательность символов, составляющая имя. Идентификатор зависит от регистра символов.

Инициализация (initialization). Присвоение переменной исходного значения при ее определении. Обычно переменные следует инициализировать.

Инициализация по умолчанию (default initialization). Способ инициализации объектов при отсутствии явной инициализации. Инициализация объектов типа класса определяется классом. Объекты встроенного типа, определенного в глобальной области видимости, инициализируются значением 0, а определенные в локальной области видимости остаются неинициализированными и имеют неопределенное значение.

Интегральный тип (integral type). То же, что и арифметический или целочисленный тип.

Ключевое слово `struct`. Используется при определении структуры (класса).

Ключевое слово `typedef`. Позволяет определить псевдоним для другого типа. Когда ключевое слово `typedef` присутствует в объявлении базового типа, определенные в объявлении имена становятся именами типа.

Константная ссылка (const reference). Разговорный термин для ссылки на константный объект.

Константное выражение (constant expression). Выражение, значение которого может быть вычислено во время компиляции.

Константный указатель (const pointer). Указатель со спецификатором `const`.

Контроль соответствия типов (type checking). Термин, описывающий процесс проверки компилятором соответствия способа использования объекта заявленному для него типу.

Литерал (literal) Значение, такое как число, символ или строка символов. Это значение не может быть изменено. Символьные литералы заключают в одинарные кавычки, а строковые литералы в двойные.

Литерал nullptr. Литеральная константа, означающая нулевой указатель.

Локальная область видимости (local scope). Разговорный синоним для области действия блока кода.

Массив (array). Структура данных, содержащая коллекцию неименованных объектов, к которым можно обращаться по индексу. Более подробная информация о массивах приведена в разделе 3.5.

Неинициализированная переменная (uninitialized variable). Переменная, определенная без исходного значения. Обычно попытка доступа к значению неинициализированной переменной приводит к неопределенному поведению.

Неопределенное поведение (undefined behavior). Случай, для которого стандарт языка не определяет значения. Осознанно или неосознанно, но полагаться на неопределенное поведение нельзя. Оно является источником трудно обнаруживаемых ошибок времени выполнения, проблем безопасности и переносимости.

Непечатаемый символ (nonprintable character). Символ, не имеющий видимого представления, например символ возврата на один символ, символ новой строки и т.д.

Нулевой указатель (null pointer). Указатель со значением 0. Нулевой указатель допустим, но не указывает ни на какой объект.

Область видимости (scope). Часть программы, в которой имена имеют смысл. Язык C++ имеет несколько уровней областей видимости.

Глобальная (global) — имена, определенные вне остальных областей видимости.

Класса (class) — имена, определенные классом.

Пространства имен (namespace) — имена, определенные в пространстве имен.

Блока (block) — имена, определенные в блоке операторов, т.е. в паре фигурных скобок.

Области видимости могут быть вложенными. Как только имя объявлено, оно доступно до конца той области видимости, в которой было объявлено.

Объект (object). Область памяти, которая имеет тип. Переменная — это объект, который имеет имя.

Объявление (declaration). Уведомление о существовании переменной, функции или типа, определяемых в другом месте программы. Никакие имена не могут быть использованы, пока они не определены или не объявлены.

Объявление псевдонима (alias declaration). Определяет синоним для другого типа. Объявление в формате `using имя = тип` объявляет *имя* как синоним типа *тип*.

Оператор&. Оператор обращения к адресу. Возвращает адрес объекта, к которому он был применен.

Оператор*. Оператор обращения к значению. Обращение к значению указателя возвращает объект, на который указывает указатель. Присвоение результату оператора обращения к значению присваивает новое значение основному объекту.

Оператор объявления (declarator). Часть объявления, включающая определяемое имя и, необязательно, модификатор типа.

Определение (definition). Резервирует область в памяти для хранения данных переменной и (необязательно) инициализирует ее значение. Никакие имена не могут быть использованы, пока они не определены или не объявлены.

Переменная (variable). Именованный объект или ссылка. В языке C++ переменные должны быть объявлены перед использованием.

Переменнаяconstexpr. Переменная, которая представляет константное выражение.

Функции `constexpr` рассматриваются в разделе 6.5.2.

Переменная препроцессора (preprocessor variable). Переменная, используемая препроцессором. Препроцессор заменяет каждую переменную препроцессора ее значением прежде, чем программа будет откомпилирована.

Переменная-член (data member). Элемент данных, которые составляют объект. Каждый объект некоего класса обладает собственными экземплярами переменных-членов. Переменные-члены могут быть инициализированы в объявлении класса.

Преобразование (conversion). Процесс, в результате которого значение одного типа преобразуется в значение другого. Преобразования между встроенными типами определены в самом языке.

Препроцессор (preprocessor). Препроцессор — это программа, автоматически запускаемая перед компилятором C++.

Псевдоним типа (type alias). Имя, являющееся синонимом для другого типа. Определяется при помощи ключевого слова `typedef` или объявления псевдонима.

Раздельная компиляция (separate compilation). Возможность разделить программу на несколько отдельных файлов исходного кода.

Связывание (*bind*). Соединение имени с указанной сущностью, чтобы использование имени приводило к использованию основной сущности. Например, ссылка — это имя, связанное с объектом.

Слово (*word*). Специфический для каждой машины размер блока памяти, применяемый при целочисленных вычислениях. Обычно размер слова достаточно велик, чтобы содержать адрес. 32-битовое слово обычно занимает 4 байта.

Составной тип (*compound type*). Тип, определенный в терминах другого типа.

Спецификатор `auto`. Спецификатор типа, позволяющий вывести тип переменной из ее инициализатора.

Спецификатор `const` **верхнего уровня** (*top-level const*). Спецификатор `const`, указывающий, что объект не может быть изменен.

Спецификатор `const` **нижнего уровня** (*low-level const*). Спецификатор `const` не верхнего уровня. Такие спецификаторы `const` являются неотъемлемой частью типа и никогда не игнорируются.

Спецификатор `const`. Спецификатор типа, определяющий объекты, которые не могут быть изменены. Константные объекты следует инициализировать, поскольку нет никакого способа присвоить им значение после определения.

Спецификатор `decltype`. Спецификатор типа, позволяющий вывести тип переменной или выражения.

Спецификатор типа (*type specifier*). Имя типа.

Списочная инициализация (*list initialization*). Форма инициализации, подразумевающая использование фигурных скобок для включения одного или нескольких инициализаторов.

Ссылка (*reference*). Псевдоним другого объекта.

Ссылка на константу (*reference to const*). Ссылка, неспособная изменить значение объекта, на который она ссылается. Ссылка на константу может быть связана с константным, неконстантным объектом или с результатом выражения.

Тип `string`. Библиотечный тип, представляющий последовательность символов переменной длины.

Тип `void*`. Специальный тип указателя, способного указывать на любой неконстантный тип. Обращение к значению таких указателей невозможно.

Тип `void`. Специальный тип без значения и допустимых операций. Нельзя определить переменную типа `void`.

Указатель (pointer). Объект, способный содержать адрес объекта, следующий адрес за концом объекта или нуль.

Указатель на константу (pointer to const). Указатель, способный содержать адрес константного объекта. Указатель на константу не может использоваться для изменения значения объекта, на который он указывает.

Управляющая последовательность (escape sequence). Альтернативный механизм представления символов. Обычно используется для представления непечатаемых символов, таких как символ новой строки или табуляции. Управляющая последовательность состоит из символа наклонной черты влево, сопровождаемой символом, восьмеричным числом из трех цифр, или символа x, сопровождаемого шестнадцатеричным числом.

Член класса (class member, member). Часть класса.

Глава 3

Типы `string`, `vector` и массивы

Кроме встроенных типов, рассмотренных в главе 2, язык C++ предоставляет богатую библиотеку абстрактных типов данных. Важнейшими библиотечными типами являются тип `string`, поддерживающий символьные строки переменной длины, и тип `vector`, определяющий коллекции переменного размера. С типами `string` и `vector` связаны типы, известные как *итераторы* (iterator). Они используются для доступа к символам строк и элементам векторов.

Типы `string` и `vector`, определенные в библиотеке, являются абстракциями более простого встроенного типа массива. Эта глава посвящена массивам и введению в библиотечные типы `vector` и `string`.

Встроенные типы, рассмотренные в главе 2, определены непосредственно языком C++. Эти типы представляют средства, которые сами по себе присущи большинству компьютеров, такие как числа или символы. Стандартная библиотека определяет множество дополнительных типов, высокоуровневый характер которых аппаратными средствами компьютеров, как правило, не реализуется непосредственно.

В данной главе представлены два важнейших библиотечных типа: `string` и `vector`. Тип `string` — это последовательность символов переменной длины. Тип `vector` содержит последовательность объектов указанного типа переменной длины. Мы также рассмотрим встроенный тип массива. Как и другие встроенные типы, массивы представляют возможности аппаратных средств. В результате массивы менее удобны в использовании, чем библиотечные типы `string` и `vector`.

Однако, прежде чем начать исследование библиотечных типов, рассмотрим механизм, упрощающий доступ к именам, определенным в библиотеке.



3.1. Пространства имен и объявления `using`

До сих пор имена из стандартной библиотеки упоминались в программах явно, т.е. перед каждым из них было указано имя пространства имен `std`. Например, при чтении со стандартного устройства ввода

применялась форма записи `std::cin`. Здесь использован *оператор области видимости ::* (см. раздел 1.2). Он означает, что имя, указанное в правом операнде оператора, следует искать в области видимости, указанной в левом операнде. Таким образом, код `std::cin` означает, что используемое имя `cin` определено в пространстве имен `std`.

При частом использовании библиотечных имен такая форма записи может оказаться чересчур громоздкой. К счастью, существуют и более простые способы применения членов пространств имен. Самый надежный из них — *объявление using* (*using declaration*). Другие способы, позволяющие упростить использование имен из других пространств, рассматриваются в разделе 18.2.2.

Объявление `using` позволяет использовать имена из другого пространства имен без указания префикса `имя_пространства_имен::`. Объявление `using` имеет следующий формат:

```
using пространство_имен::имя;
```

После того как объявление `using` было сделано один раз, к указанному в нем имени можно обращаться без указания пространства имен.

```
#include <iostream>
// объявление using; при использовании имени cin
теперь
// подразумевается, что оно принадлежит
пространству имен std
using std::cin;
int main() {
    int i;
    cin >> i;           // ok: теперь cin - синоним
std::cin
    cout << i;          // ошибка: объявления using нет;
здесь нужно указать
                    // полное имя
    std::cout << i; // ok: явно указано применение
cout из
                    // пространства имен std
    return 0;
}
```

*Для каждого имени необходимо индивидуальное объявление
using*

Каждое объявление `using` применяется только к одному элементу пространства имен. Это позволяет жестко задавать имена, используемые в каждой программе. Например, программу из раздела 1.2 можно переписать следующим образом:

```
#include <iostream>
// объявления using для имен из стандартной
библиотеки
using std::cin;
using std::cout;
using std::endl;
int main() {
    cout << "Enter two numbers:" << endl;
    int v1, v2;
    cin >> v1 >> v2;
    cout << "The sum of " << v1 << " and " << v2
        << " is " << v1 + v2 << endl;
    return 0;
}
```

Объявления `using` для имен `cin`, `cout` и `endl` означают, что их можно теперь использовать без префикса `std::`. Напомню, что программы C++ позволяют поместить каждое объявление `using` в отдельную строку или объединить в одной строке несколько объявлений. Важно не забывать, что для каждого используемого имени необходимо отдельное объявление `using`, и каждое из них должно завершаться точкой с запятой.

Заголовки не должны содержать объявлений `using`

Код в заголовках (см. раздел 2.6.3) обычно не должен использовать объявления `using`. Дело в том, что содержимое заголовка копируется в текст программы, в которую он включен. Если в заголовке есть объявление `using`, то каждая включающая его программа получает то же объявление `using`. В результате программа, которая не намеревалась использовать определенное библиотечное имя, может случайно столкнуться с неожиданным конфликтом имен.

Примечание для читателя

Начиная с этого момента подразумевается, что во все примеры включены объявления `using` для имен из стандартной библиотеки. Таким

образом, в тексте и примерах кода далее упоминается `cin`, а не `std::cin`.

Кроме того, для экономии места в примерах кода не будем показывать далее объявления `using` и необходимые директивы `#include`. В табл. А.1 приложения А приведены имена и соответствующие заголовки стандартной библиотеки, которые использованы в этой книге.



Читатели не должны забывать добавить соответствующие объявления `#include` и `using` в свои примеры перед их компиляцией.

Упражнения раздела 3.1

Упражнение 3.1. Перепишите упражнения из разделов 1.4.1 и 2.6.2, используя соответствующие объявления `using`.



3.2. Библиотечный тип `string`

Строка (`string`) — это последовательность символов переменной длины. Чтобы использовать тип `string`, необходимо включить в код заголовок `string`. Поскольку тип `string` принадлежит библиотеке, он определен в пространстве имен `std`. Наши примеры подразумевают наличие следующего кода:

```
#include <string>
using std::string;
```

В этом разделе описаны наиболее распространенные операции со строками; а дополнительные операции рассматриваются в разделе 9.5.



Кроме определения операций, предоставляемых библиотечными типами, стандарт налагает также требования на эффективность их конструкторов. В результате библиотечные типы оказались весьма эффективны в использовании.



3.2.1. Определение и инициализация строк

Каждый класс определяет, как могут быть инициализированы объекты его типа. Класс может определить много разных способов инициализации объектов своего типа. Каждый из способов отличается либо количеством предоставляемых инициализаторов, либо типами этих инициализаторов. Список наиболее распространенных способов инициализации строк приведен в табл. 3.1, а некоторые из примеров приведены ниже.

```
string s1;           // инициализация по умолчанию;
s1 - пустая строка
string s2 = s1;      // s2 - копия s1
string s3 = "hiya";   // s3 - копия строкового
литерала
```

```
string s4( 10, 'c' ); // s4 - cccccccccc
```

Инициализация строки по умолчанию (см. раздел 2.2.1) создает пустую строку; т.е. объект класса `string` без символов. Когда предоставляется строковый литерал (см. раздел 2.1.3), во вновь созданную строку копируются символы этого литерала, исключая завершающий нулевой символ. При предоставлении количества и символа строка содержит указанное количество экземпляров данного символа.

Таблица 3.1. Способы инициализации объекта класса `string`

<code>string s1</code>	Инициализация по умолчанию; <code>s1</code> — пустая строка
<code>string s2(s1)</code>	<code>s2</code> — копия <code>s1</code>
<code>string s2 = s1</code>	Эквивалент <code>s2(s1)</code> , <code>s2</code> — копия <code>s1</code>
<code>string s3("value")</code>	<code>s3</code> — копия строкового литерала, нулевой символ не включен
<code>string s3 = "value"</code>	Эквивалент <code>s3("value")</code> , <code>s3</code> — копия строкового литерала
<code>string s4(n, 'c')</code>	Инициализация переменной <code>s4</code> символом 'c' в количестве <code>n</code> штук

Прямая инициализация и инициализация копией

В разделе 2.2.1 упоминалось, что язык C++ поддерживает несколько разных форм инициализации. Давайте на примере класса `string` начнем изучать, чем эти формы отличаются друг от друга. Когда переменная инициализируется с использованием знака `=`, компилятор просит скопировать инициализирующий объект в создаваемый объект, т.е. выполнить *инициализацию копией* (*copy initialization*). В противном случае без знака `=` осуществляется *прямая инициализация* (*direct initialization*).

Когда имеется одиночный инициализатор, можно использовать и прямую форму, и инициализацию копией. Но при инициализации переменной несколькими значениями, как при инициализации переменной `s4` выше, следует использовать прямую форму инициализации.

```
string s5 = "hiya"; // инициализация копией
string s6( "hiya" ); // прямая инициализация
string s7( 10, 'c' ); // прямая инициализация; s7 -
                     - cccccccccc
```

Если необходимо использовать несколько значений, можно применить косвенную форму инициализации копией при явном создании временного объекта для копирования.

```
string s8 = string( 10, 'c' ); // инициализация
```

копией; s8 – cccccccccc

Инициализатор строки `s8 = string(10, 'c')` — создает строку заданного размера, заполненную указанным символьным значением, а затем копирует ее в строку `s8`. Это эквивалентно следующему коду:

```
string temp(10, 'c'); // temp - cccccccccc  
string s8 = temp; // копировать temp в s8
```

Хотя используемый для инициализации строки `s8` код вполне допустим, он менее читабелен и не имеет никаких преимуществ перед способом, которым была инициализирована переменная `s7`.



3.2.2. Операции со строками

Наряду с определением способов создания и инициализации объектов класс определяет также операции, которые можно выполнять с объектами класса. Класс может определить обладающие именем операции, такие как функция `isbn()` класса `Sales_item` (см. раздел 1.5.2). Класс также может определить то, что означают различные символы операторов, такие как `<<` или `+`, когда они применяются к объектам класса. Наиболее распространенные операции класса `string` приведены в табл. 3.2.

Таблица 3.2. Операции класса `string`

<code>os << s</code>	Выводит строку <code>s</code> в поток вывода <code>os</code> . Возвращает поток <code>os</code>
<code>is >> s</code>	Читает разделенную пробелами строку <code>s</code> из потока <code>is</code> . Возвращает поток <code>is</code>
<code>getline(is, s)</code>	Читает строку ввода из потока <code>is</code> в переменную <code>s</code> . Возвращает поток <code>is</code>
<code>s.empty()</code>	Возвращает значение <code>true</code> , если строка <code>s</code> пуста. В противном случае возвращает значение <code>false</code>
<code>s.size()</code>	Возвращает количество символов в строке <code>s</code>
<code>s[n]</code>	Возвращает ссылку на символ в позиции <code>n</code> строки <code>s</code> ; позиции отчитываются от 0
<code>s1 + s2</code>	Возвращает строку, состоящую из содержимого строк <code>s1</code> и <code>s2</code>
<code>s1 = s2</code>	Заменяет символы строки <code>s1</code> копией содержимого строки <code>s2</code>
<code>s1 == s2</code> <code>s1 != s2</code>	Строки <code>s1</code> и <code>s2</code> равны, если содержат одинаковые символы. Регистр символов учитывается
<code><, <=, >, >=</code>	Сравнение зависит от регистра и полагается на алфавитный порядок символов

Чтение и запись строк

Как уже упоминалось в главе 1, для чтения и записи значений встроенных типов, таких как `int`, `double` и т.д., используется библиотека `iostream`. Для чтения и записи строк используются те же операторы ввода и вывода.

// Обратите внимание: перед компиляцией этот код

следует дополнить

```
// директивами #include и объявлениями using
int main() {
    string s;           // пустая строка
    cin >> s;          // чтение разделяемой пробелами
 строки в s
    cout << s << endl; // запись s в поток вывода
    return 0;
}
```

Программа начинается с определения пустой строки по имени `s`. Следующая строка читает данные со стандартного устройства ввода и сохраняет их в переменной `s`. Оператор ввода строк читает и отбрасывает все предваряющие непечатаемые символы (например, пробелы, символы новой строки и табуляции). Затем он читает значащие символы, пока не встретится следующий непечатаемый символ.

Таким образом, если ввести "**Hello World!**" (обратите внимание на предваряющие и завершающие пробелы), фактически будет получено значение "Hello" без пробелов.

Подобно операторам ввода и вывода встроенных типов, операторы строк возвращают как результат свой левый operand. Таким образом, операторы чтения или записи можно объединять в цепочки.

```
string s1, s2;
cin >> s1 >> s2; // сначала прочитать в переменную
s1,                      // а затем в переменную s2
cout << s1 << s2 << endl; // отобразить обе строки
```

Если в этой версии программы осуществить предыдущий ввод, "**Hello World!**", выводом будет "HelloWorld!" .

Чтение неопределенного количества строк

В разделе 1.4.3 уже рассматривалась программа, читающая неопределенное количество значений типа `int`. Напишем подобную программу, но читающую строки.

```
int main() {
    string word;
    while (cin >> word) // читать до конца файла
        cout << word << endl; // отобразить каждое слово
 с новой строки
```

```
    return 0;
}
```

Здесь чтение осуществляется в переменную типа `string`, а не `int`. Условие оператора `while`, напротив, выполняется так же, как в предыдущей программе. Условие проверяет поток после завершения чтения. Если поток допустим, т.е. не встретился символ конца файла или недопустимое значение, выполняется тело цикла `while`. Оно выводит прочитанное значение на стандартное устройство вывода. Как только встречается конец файла (или недопустимый ввод), цикл `while` завершается.

Применение функции `getline()` для чтения целой строки

Иногда игнорировать пробелы во вводе не нужно. В таких случаях вместо оператора `>>` следует использовать функцию `getline()`. Функция `getline()` получает поток ввода и строку. Функция читает предоставленный поток до первого символа новой строки и сохраняет прочитанное, исключая символ новой строки, в своем аргументе типа `string`. Встретив символ новой строки, даже если это первый символ во вводе, функция `getline()` прекращает чтение и завершает работу. Если символ новой строки во вводе первый, то возвращается пустая строка.

Подобно оператору ввода, функция `getline()` возвращает свой аргумент типа `istream`. В результате функцию `getline()` можно использовать в условии, как и оператор ввода (см. раздел 1.4.3). Например, предыдущую программу, которая выводила по одному слову в строку, можно переписать так, чтобы она вместо этого выводила всю строку:

```
int main() {
    string line;
    // читать строки до конца файла
    while (getline(cin, line))
        cout << line << endl;
    return 0;
}
```

Поскольку переменная `line` не будет содержать символа новой строки, его придется вывести отдельно. Для этого, как обычно, используется манипулятор `endl`, который, кроме перевода строки, сбрасывает буфер вывода.



Символ новой строки, прекращающий работу функции `getline()`, отбрасывается и в строковой переменной *не сохраняется*.

Строковые операции `size()` и `empty()`

Функция `empty()` (пусто) делает то, что и ожидается: она возвращает логическое значение `true` (раздел 2.1), если строка пуста, и значение `false` — в противном случае. Подобно функции-члену `isbn()` класса `Sales_item` (см. раздел 1.5.2), функция `empty()` является членом класса `string`. Для вызова этой функции используем точечный оператор, позволяющий указать объект, функцию `empty()` которого необходимо вызвать.

А теперь пересмотрим предыдущую программу так, чтобы она выводила только непустые строки:

```
// читать ввод построчно и отбрасывать пустые строки
while (getline(cin, line))
    if (!line.empty())
        cout << line << endl;
```

Условие использует *оператор логического NOT* (*оператор !*). Он возвращает инверсное значение своего операнда типа `bool`. В данном случае условие истинно, если строка `line` не пуста.

Функция `size()` возвращает длину строки (т.е. количество символов в ней). Давайте используем ее для вывода строк длиной только больше 80 символов.

```
string line;
// читать ввод построчно и отображать строки длиной более 80 символов
while (getline(cin, line))
    if (line.size() > 80)
        cout << line << endl;
```

Tun `string`::`size_type`

Вполне логично ожидать, что функция `size()` возвращает значение типа `int`, а учитывая сказанное в разделе 2.1.1, вероятней всего, типа

`unsigned`. Но вместо этого функция `size()` возвращает значение типа `string::size_type`. Этот тип требует более подробных объяснений.

В классе `string` (и нескольких других библиотечных типах) определены вспомогательные типы данных. Эти вспомогательные типы позволяют использовать библиотечные типы машинно-независимым способом. Тип `size_type` — это один из таких вспомогательных типов. Чтобы воспользоваться типом `size_type`, определенным в классе `string`, применяется оператор области видимости (оператор `::`), указывающий на то, что имя `size_type` определено в классе `string`.

Хотя точный размер типа `string::size_type` неизвестен, можно с уверенностью сказать, что этот беззнаковый тип (см. раздел 2.1.1) достаточно большой, чтобы содержать размер любой строки. Любая переменная, используемая для хранения результата операции `size()` класса `string`, должна иметь тип `string::size_type`.



По общему признанию, довольно утомительно вводить каждый раз тип `string::size_type`. По новому стандарту можно попросить компилятор самостоятельно применить соответствующий тип при помощи спецификаторов `auto` или `decltype` (см. раздел 2.5.2):

```
auto len = line.size(); // len имеет тип  
string::size_type
```

Поскольку функция `size()` возвращает беззнаковый тип, следует напомнить, что выражения, в которых смешаны знаковые и беззнаковые данные, могут дать непредвиденные результаты (см. раздел 2.1.2). Например, если переменная `n` типа `int` содержит отрицательное значение, то выражение `s.size() < n` почти наверняка истинно. Оно возвращает значение `true` потому, что отрицательное значение переменной `n` преобразуется в большое беззнаковое значение.



Проблем преобразования между беззнаковыми и знаковыми типами можно избежать, если не использовать переменные типа `int` в выражениях, где используется функция `size()`.

Сравнение строк

Класс `string` определяет несколько операторов для сравнения строк. Эти операторы сравнивают строки посимвольно. Результат сравнения зависит от регистра символов, символы в верхнем и нижнем регистре отличаются.

Операторы равенства (`==` и `!=`) проверяют, равны или не равны две строки соответственно. Две строки равны, если у них одинаковая длина и одинаковые символы. Операторы сравнения (`<`, `>`, `<=`, `>=`) проверяют, меньше ли одна строка другой, больше, меньше или равна, больше или равна другой. Эти операторы используют ту же стратегию, старшинство символов в алфавитном порядке в зависимости от регистра.

1. Если длина у двух строк разная и если каждый символ более короткой строки совпадает с соответствующим символом более длинной, то короткая строка меньше длинной.

2. Если символы в соответствующих позициях двух строк отличаются, то результат сравнения определяется первым отличающимся символом.

Для примера рассмотрим следующие строки:

```
string str = "Hello";
string phrase = "Hello World";
string slang = "Hiya";
```

Согласно правилу 1 строка `str` меньше строки `phrase`. Согласно правилу 2 строка `slang` больше, чем строки `str` и `phrase`.

Присвоение строк

Как правило, библиотечные типы столь же просты в применении, как и встроенные. Поэтому большинство библиотечных типов поддерживают присвоение. Строки не являются исключением, один объект класса `string` вполне можно присвоить другому.

```
string st1(10, 'c'), st2; // st1 - cccccccccc; st2
- пустая строка
st1 = st2; // присвоение: замена содержимого st1
копией st2
// теперь st1 и st2 - пустые строки
```

Сложение двух строк

Результатом сложения двух строк является новая строка, объединяющая содержимое левого операнда, а затем правого. Таким образом, при применении *оператора суммы* (оператор `+`) к строкам результатом будет новая строка, символы которой являются копией символов левого операнда, сопровождаемые символами правого операнда.

Составной оператор присвоения (оператор `+=`) (см. раздел 1.4.1) добавляет правый операнд к строке слева:

```
string s1 = "hello, ", s2 = "world\n";
string s3 = s1 + s2; // s3 - hello, world\n
s1 += s2;           // эквивалентно s1 = s1 + s2
```

Сложение строк и символьных строковых литералов

Как уже упоминалось в разделе 2.1.2, один тип можно использовать там, где ожидается другой тип, если есть преобразование из данного типа в ожидаемый. Библиотека `string` позволяет преобразовывать как символьные, так и строковые литералы (см. раздел 2.1.3) в строки. Поскольку эти литералы можно использовать там, где ожидаются строки, предыдущую программу можно переписать следующим образом:

```
string s1 = "hello", s2 = "world"; // в s1 и s2 нет
пунктуации
```

```
string s3 = s1 + ", " + s2 + '\n';
```

Когда объекты класса `string` смешиваются со строковыми или символьными литералами, то по крайней мере один из операндов каждого оператора `+` должен иметь тип `string`.

```
string s4 = s1 + ", ";                  // ok: сложение
строки и литерала
```

```
string s5 = "hello" + ", ";             // ошибка: нет
строкового операнда
```

```
string s6 = s1 + ", " + "world"; // ok: каждый +
имеет
```

```
                                // строковый
операнд
```

```
string s7 = "hello" + ", " + s2; // ошибка: нельзя
сложить строковые
```

```
                                // литералы
```

В инициализации переменных `s4` и `s6` задействовано только по одному оператору, поэтому достаточно просто проверить его корректность. Инициализация переменной `s6` может показаться странной, но работает она аналогично объединенным в цепочку операторам ввода или вывода (см. раздел 1.2). Это эквивалентно следующему коду:

```
string s6 = (s1 + ", ") + "world";
```

Часть `s1 + ", "` выражения возвращает объект класса `string`, она составляет левый операнд второго оператора `+`. Это эквивалентно

следующему коду:

```
string tmp = s1 + ", " ; // ok: + имеет строковый
операнд
s6 = tmp + "world";           // ok: + имеет строковый
операнд
```

С другой стороны, инициализация переменной `s7` недопустима, и это становится очевидным, если заключить часть выражения в скобки:

```
string s7 = ("hello" + ", ") + s2; // ошибка:
нельзя сложить строковые
                                         // литералы
```

Теперь довольно просто заметить, что первая часть выражения суммирует два строковых литерала. Поскольку это невозможно, оператор недопустим.



По историческим причинам и для совместимости с языком С строковые литералы *не принадлежат* к типу `string` стандартной библиотеки. При использовании строковых литералов и библиотечного типа `string`, не следует забывать, что это разные типы.

Упражнения раздела 3.2.2

Упражнение 3.2. Напишите программу, читающую со стандартного устройства ввода по одной строке за раз. Измените программу так, чтобы читать по одному слову за раз.

Упражнение 3.3. Объясните, как символы пробелов обрабатываются в операторе ввода класса `string` и в функции `getline()`.

Упражнение 3.4. Напишите программу, читающую две строки и сообщающую, равны ли они. В противном случае программа сообщает, которая из них больше. Затем измените программу так, чтобы она сообщала, одинаковая ли у строк длина, а в противном случае — которая из них длиннее.

Упражнение 3.5. Напишите программу, читающую строки со стандартного устройства ввода и суммирующую их в одну большую строку. Отобразите полученную строку. Затем измените программу так, чтобы отделять соседние введенные строки пробелами.



3.2.3. Работа с символами строки

Зачастую приходится работать с индивидуальными символами строки. Например, может понадобиться выяснить, является ли определенный символ пробелом, или изменить регистр символов на нижний, или узнать, присутствует ли некий символ в строке, и т.д.

Одной из частей этих действий является доступ к самим символам строки. Иногда необходима обработка каждого символа, а иногда лишь определенного символа, либо может понадобиться прекратить обработку, как только выполнится некое условие. Кроме того, это наилучший способ справиться со случаями, когда действуются разные языковые и библиотечные средства.

Другой частью обработки символов является выяснение и (или) изменение их характеристик. Эта часть задачи выполняется набором библиотечных функций, описанных в табл. 3.3. Данные функции определены в заголовке `cctype`.

Таблица 3.3. Функции `cctype`

<code>isalnum(c)</code>	Возвращает значение <code>true</code> , если <code>c</code> является буквой или цифрой
<code>isalpha(c)</code>	Возвращает значение <code>true</code> , если <code>c</code> — буква
<code>iscntrl(c)</code>	Возвращает значение <code>true</code> , если <code>c</code> — управляемый символ
<code>isdigit(c)</code>	Возвращает значение <code>true</code> , если <code>c</code> — цифра
<code>isgraph(c)</code>	Возвращает значение <code>true</code> , если <code>c</code> — не пробел, а печатаемый символ
<code>islower(c)</code>	Возвращает значение <code>true</code> , если <code>c</code> — символ в нижнем регистре
<code>isprint(c)</code>	Возвращает значение <code>true</code> , если <code>c</code> — печатаемый символ
<code>ispunct(c)</code>	Возвращает значение <code>true</code> , если <code>c</code> — знак пунктуации (т.е. символ, который не является управляемым символом, цифрой, символом или печатаемым отступом)
<code>isspace(c)</code>	Возвращает значение <code>true</code> , если <code>c</code> — символ отступа (т.е. пробел, табуляция, вертикальная табуляция, возврат, новая строка или прогон страницы)
<code>isupper(c)</code>	Возвращает значение <code>true</code> , если <code>c</code> — символ в верхнем регистре
<code>isxdigit(c)</code>	Возвращает значение <code>true</code> , если <code>c</code> — шестнадцатеричная цифра
	Если <code>c</code> — прописная буква, возвращает ее эквивалент в нижнем

<code>tolower(c)</code>	регистре, в противном случае возвращает символ с неизменным
<code>toupper(c)</code>	Если <code>c</code> — строчная буква, возвращает ее эквивалент в верхнем регистре, в противном случае возвращает символ с неизменным

Совет. Используйте версии C++ библиотечных заголовков языка С

Кроме средств, определенных специально для языка C++, его библиотека содержит также библиотеку языка С. Имена заголовков языка С имеют формат *имя. h*. Версии этих же заголовков языка C++ имеют формат *симя*, т.е. суффикс *. h* удален, а *имени* предшествует символ *c*, означающий, что этот заголовок принадлежит библиотеке С.

Следовательно, у заголовка *cctype* то же содержимое, что и у заголовка *ctype. h*, но в форме, соответствующей программе C++. В частности, имена, определенные в заголовках *с имя*, определены также в пространстве имен *std*, тогда как имена, определенные в заголовках *. h*, — нет.

Как правило, в программах на языке C++ используют заголовки версии *симя*, а не *имя. h*. Таким образом, имена из стандартной библиотеки будут быстро найдены в пространстве имен *std*. Использование заголовка *. h* возлагает на программиста дополнительную заботу по отслеживанию, какие из библиотечных имен унаследованы от языка С, а какие принадлежат языку C++.

Обработка каждого символа, использование серийного оператора for



Если необходимо сделать нечто с каждым символом в строке, то наилучшим подходом является использование оператора, введенного новым стандартом, — *серийный оператор for* (*range for*). Этот оператор перебирает элементы данной ему последовательности и выполняет с каждым из них некую операцию. Его синтаксическая форма такова:

```
for ( объявление : выражение)
    оператор
```

где *выражение* — это объект типа, который представляет последовательность, а *объявление* определяет переменную, которая будет использована для доступа к элементам последовательности. На

каждой итерации переменная в *объявлении* инициализируется значением следующего элемента в *выражении*.

Строка представляет собой последовательность символов, поэтому объект типа `string` можно использовать как *выражение* в серийном операторе `for`. Например, серийный оператор `for` можно использовать для вывода каждого символа строки в отдельной строке вывода.

```
string str( "some string" );
// вывести символы строки str по одному на строку
for (auto c : str) // для каждого символа в строке
str
    cout << c << endl; // вывести текущий символ и
имя символ новой строки
```

Цикл `for` ассоциирует переменную `c` с переменной `str` типа `string`. Управляющая переменная цикла определяется тем же способом, что и любая другая переменная. В данном случае используется спецификатор `auto` (см. раздел 2.5.2), чтобы позволить компилятору самостоятельно определять тип переменной `c`, которым в данном случае будет тип `char`. На каждой итерации следующий символ строки `str` будет скопирован в переменную `c`. Таким образом, можно прочитать этот цикл так: "Для каждого символа `c` в строке `str`" сделать нечто. Под "нечто" в данном случае подразумевается вывод текущего символа, сопровождаемого символом новой строки.

Рассмотрим более сложный пример и используем серийный оператор `for`, а также функцию `ispunct()` для подсчета количества знаков пунктуации в строке:

```
string s("Hello World! !! ");
// punct_cnt имеет тот же тип, что и у
возвращаемого значения
// функции s.size(); см. р. 2.5.3
decltype(s.size()) punct_cnt = 0;
// подсчитать количество знаков пунктуации в строке
s
for (auto c : s) // для каждого символа в строке s
    if (ispunct(c)) // если символ знак пунктуации
        ++punct_cnt; // увеличить счетчик пунктуаций
cout << punct_cnt
    << " punctuation characters in " << s << endl;
```

Вывод этой программы таков:

3 punctuation characters in Hello World! !!

Здесь для объявления счетчика `punct_cnt` используется спецификатор `decltype` (см. раздел 2.5.3). Его тип совпадает с типом возвращаемого значения функции `s.size()`, которым является тип `string::size_type`. Для обработки каждого символа в строке используем серийный оператор `for`. На сей раз проверяется, является ли каждый символ знаком пунктуации. Если да, то используем оператор инкремента (см. раздел 1.4.1) для добавления единицы к счетчику. Когда серийный оператор `for` завершает работу, отображается результат.

Использование серийного оператора `for` для изменения символов в строке

Если необходимо изменить значение символов в строке, переменную цикла следует определить как ссылочный тип (см. раздел 2.3.1). Помните, что ссылка — это только другое имя для данного объекта. При использовании ссылки в качестве управляющей переменной она будет поочереди связана с каждым элементом последовательности. Используя ссылку, можно изменить символ, с которым она связана.

Предположим, что вместо подсчета знаков пунктуации необходимо преобразовать все буквы строки в верхний регистр. Для этого можно использовать библиотечную функцию `toupper()`, которая возвращает полученный символ в верхнем регистре. Для преобразования всей строки необходимо вызвать функцию `toupper()` для каждого символа и записать результат в тот же символ:

```
string s("Hello World! !!");
// преобразовать s в верхний регистр
for (auto &c : s) // для каждого символа в строке s
    // (примечание: c - ссылка)
    c = toupper(c); // c - ссылка, поэтому присвоение
изменяет
    // символ в строке s
cout << s << endl;
```

Вывод этого кода таков:

HELLO WORLD! !!

На каждой итерации переменная `c` ссылается на следующий символ строки `s`. При присвоении значения переменной `c` изменяется соответствующий символ в строке `s`.

```
c = toupper(c); // c - ссылка, поэтому присвоение
```

изменяет

```
// символ в строке s
```

Таким образом, данное выражение изменяет значение символа, с которым связана переменная `s`. По завершении этого цикла все символы в строке `str` будут в верхнем регистре.

Обработка лишь некоторых символов

Серийный оператор `for` работает хорошо, когда необходимо обработать каждый символ. Но иногда необходим доступ только к одному символу или к некоторому количеству символов на основании некоего условия. Например, можно преобразовать в верхний регистр только первый символ строки или только первое слово в строке.

Существуют два способа доступа к отдельным символам в строке: можно использовать индексирование или итератор. Более подробная информация об итераторах приведена в разделе 3.4 и в главе 9.

Оператор индексирования (оператор []) получает значение типа `string::size_type` (раздел 3.2.2), обозначающее позицию символа, к которому необходим доступ. Оператор возвращает ссылку на символ в указанной позиции.

Индексация строк начинается с нуля; если строка `s` содержит по крайней мере два символа, то первым будет символ `s[0]`, вторым — `s[1]`, а последним символом является `s[s.size() - 1]`.



Значения, используемые для индексирования строк, не должны быть отрицательными и не должны превосходить размер строки (≥ 0 и $< \text{size}()$). Результат использования индекса вне этого диапазона непредсказуем. Непредсказуема также индексация пустой строки.

Значение оператора индексирования называется *индексом* (index). Индекс может быть любым выражением, возвращающим целочисленное значение. Если у индекса будет знаковый тип, то его значение преобразуется в беззнаковый тип `size_type` (см. раздел 2.1.2).

Следующий пример использует оператор индексирования для вывода первого символа строки:

```
if ( ! s.empty( ) ) // удостоверившись, что
```

```
символ для вывода есть,  
cout << s[0] << endl; // вывести первый символ  
строки s
```

Прежде чем обратиться к символу, удостоверимся, что строка *s* не пуста. При каждом использовании индексирования следует проверять наличие значения в данной области. Если строка *s* пуста, то значение *s[0]* неопределено.

Если строка не константа (см. раздел 2.4), возвращенному оператором индексирования символу можно присвоить новое значение. Например, первый символ можно перевести в верхний регистр следующим образом:

```
string s("some string");  
if (!s.empty()) // удостовериться в наличии  
символа s[0]  
    s[0] = toupper(s[0]); // присвоить новое значение  
первому символу
```

Вывод этой программы приведен ниже.

```
Some string
```

Использование индексирования для перебора

В следующем примере переведем в верхний регистр первое слово строки *s*:

```
// обрабатывать символы строки s, пока они не  
исчертятся или  
// не встретится пробел  
for (decltype(s.size()) index = 0;  
    index != s.size() && !isspace(s[index]); ++index)  
    s[index] = toupper(s[index]); // преобразовать в  
верхний регистр
```

Вывод этой программы таков:

```
SOME string
```

Цикл *for* (см. раздел 1.4.2) использует переменную *index* для индексирования строки *s*. Для присвоения переменной *index* соответствующего типа используется спецификатор *decltype*. Переменную *index* инициализируем значением 0, чтобы первая итерация началась с первого символа строки *s*. На каждой итерации значение переменной *index* увеличивается, чтобы получить следующий символ строки *s*. В теле цикла текущий символ переводится в верхний регистр.

В условии цикла *for* используется новая часть — *оператор*

логического *AND* (*оператор &&*). Этот оператор возвращает значение `true`, если оба операнда истинны, и значение `false` в противном случае. Важно то, что этот оператор гарантирует обработку своего правого операнда, *только если* левый операнд истинен. В данном случае это гарантирует, что индексирования строки `s` не будет, если переменная `index` находится вне диапазона. Таким образом, часть `s[index]` выполняется, только если переменная `index` не равна `s.size()`. Поскольку инкремент переменной `index` никогда не превзойдет значения `s.size()`, переменная `index` всегда будет меньше `s.size()`.

Внимание! Индексирование не контролируется

При использовании индексирования следует самому позаботиться о том, чтобы индекс оставался в допустимом диапазоне. Индекс должен быть ≥ 0 и $< \text{size}()$ строки. Для упрощения кода, использующего индексирование, в качестве индекса *всегда* следует использовать переменную типа `string::size_type`. Поскольку это беззнаковый тип, индекс не может быть меньше нуля. При использовании значения типа `size_type` в качестве индекса достаточно проверять только то, что значение индекса меньше значения, возвращаемого функцией `size()`.



Библиотека не обязана проверять и не проверяет значение индекса. Результат использования индекса вне диапазона непредсказуем.

Использование индексирования для произвольного доступа

В предыдущем примере преобразования регистра символов последовательности индекс перемещался на одну позицию за раз. Но можно также вычислить индекс и непосредственно обратиться к выбранному символу. Нет никакой необходимости получать доступ к символам последовательно.

Предположим, например, что имеется число от 0 до 15, которое необходимо представить в шестнадцатеричном виде. Для этого можно использовать строку, инициализированную шестнадцатью шестнадцатеричными цифрами.

```
const string hexdigits = "0123456789ABCDEF"; // возможные
```

```
//  
шестнадцатеричные цифры  
cout << "Enter a series of numbers between 0 and  
15"  
        << " separated by spaces. Hit ENTER when  
finished: "  
        << endl;  
string result; // будет содержать  
результатирующую // шестнадцатеричную строку  
string::size_type n; // содержит введенное число  
while (cin >> n)  
    if (n < hexdigits.size()) // игнорировать  
недопустимый ввод  
    result += hexdigits[n]; // выбрать указанную  
// шестнадцатеричную  
цифру
```

```
cout << "Your hex number is: " << result << endl;
```

Если ввести следующие числа:

12 0 5 15 8 15

то результат будет таким:

```
Your hex number is: C05F8F
```

Программа начинается с инициализации строки `hexdigits`, содержащей шестнадцатеричные цифры от 0 до F. Сделаем эту строку константной (см. раздел 2.4), поскольку содержащиеся в ней значения не должны изменяться. Для индексирования строки `hexdigits` используем в цикле введенное значение `n`. Значением `hexdigits[n]` является символ, расположенный в позиции `n` строки `hexdigits`. Например, если `n` равно 15, то результат — F; если 12, то результат — C и т.д. Полученная цифра добавляется к переменной `result`, которая и выводится, когда весь ввод прочитан.

Всякий раз, когда используется индексирование, следует позаботиться о том, чтобы индекс оставался в диапазоне. В этой программе индекс, `n`, имеет тип `string::size_type`, который, как известно, является беззнаковым. В результате значение переменной `n` гарантированно будет больше или равно 0. Прежде чем использовать переменную `n` для индексирования строки `hexdigits`, удостоверимся, что ее значение меньше, чем `hexdigits.size()`.

Упражнения раздела 3.2.3

Упражнение 3.6. Используйте серийный оператор `for` для замены всех символов строки на X.

Упражнение 3.7. Что будет, если определить управляющую переменную цикла в предыдущем упражнении как имеющую тип `char`? Предскажите результат, а затем измените программу так, чтобы использовался тип `char`, и убедитесь в своей правоте.

Упражнение 3.8. Перепишите программу первого упражнения, сначала используя оператор `while`, а затем традиционный цикл `for`. Какой из этих трех подходов вы предпочтете и почему?

Упражнение 3.9. Что делает следующая программа? Действительно ли она корректна? Если нет, то почему?

```
string s;  
cout << s[0] << endl;
```

Упражнение 3.10. Напишите программу, которая читает строку символов, включающую знаки пунктуации, и выведите ее, но уже без знаков пунктуации.

Упражнение 3.11. Допустим ли следующий серийный оператор `for`? Если да, то каков тип переменной `c`?

```
const string s = "Keep out!";  
for (auto &c : s) { /*...*/}
```



3.3. Библиотечный тип `vector`

Вектор (`vector`) — это коллекция объектов одинакового типа, каждому из которых присвоен целочисленный индекс, предоставляющий доступ к этому объекту. Вектор — это *контейнер* (*container*), поскольку он "содержит" другие объекты. Более подробная информация о контейнерах приведена в части II.

Чтобы использовать вектор, необходимо включить соответствующий заголовок. В примерах подразумевается также, что включено соответствующее объявление `using`.

```
#include <vector>
using std::vector;
```

Tinvector — это *шаблон класса* (*class template*). Язык C++ поддерживает шаблоны и классов, и функций. Написание шаблона требует довольно глубокого понимания языка C++. До главы 16 мы даже не будем рассматривать создание собственных шаблонов! К счастью, чтобы использовать шаблоны, вовсе не обязательно уметь их создавать.

Шаблоны сами по себе не являются ни функциями, ни классами. Их можно считать инструкцией для компилятора по созданию классов или функций. Процесс, используемый компилятором для создания классов или функций по шаблону, называется *созданием экземпляра* (*instantiation*) шаблона. При использовании шаблона необходимо указать, экземпляр какого класса или функции должен создать компилятор.

Для создания экземпляра шаблона класса следует указать дополнительную информацию, характер которой зависит от шаблона. Эта информация всегда задается одинаково: в угловых скобках после имени шаблона.

В случае вектора предоставляемой дополнительной информацией является тип объектов, которые он должен содержать:

```
vector<int> ivec;                                // ivec содержит
объекты типа int
vector<Sales_item> Sales_vec; // содержит объекты
класса Sales_item
vector<vector<string>> file; // вектор, содержащий
другие векторы
```

В этом примере компилятор создает три разных экземпляра шаблона `vector`: `vector<int>`, `vector<Sales_item>` и `vector<vector<string>>`.



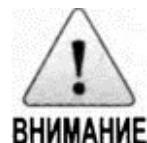
`vector` — это шаблон, а не класс. Классам, созданным по шаблону `vector`, следует указать тип хранимого элемента, например `vector<int>`.

Можно определить векторы для содержания объектов практически любого типа. Поскольку ссылки не объекты (см. раздел 2.3.1), не может быть вектора ссылок. Однако векторы большинства других (не ссылочных) встроенных типов и типов классов вполне могут существовать. В частности, может быть вектор, элементами которого являются другие векторы.



Следует заметить, что прежние версии языка C++ использовали несколько иной синтаксис определения вектора, элементы которого сами являлись экземплярами шаблона `vector` (или другого типа шаблона). Прежде необходимо было ставить пробел между закрывающей угловой скобкой внешней части `vector` и типом его элемента: т.е.

`vector<vector<int> >`, а не `vector<vector<int>>`.



ВНИМАНИЕ

Некоторые компиляторы могут потребовать объявления вектора векторов в старом стиле, например `vector<vector<int> >`.



3.3.1. Определение и инициализация векторов

Подобно любому типу класса, шаблон `vector` контролирует способ определения и инициализации векторов. Наиболее распространенные способы определения векторов приведены в табл. 3.4.

Инициализация вектора по умолчанию (см. раздел 2.2.1) позволяет создать пустой вектор определенного типа:

```
vector<string> svec; // инициализация по умолчанию;  
                      // у svec нет элементов
```

Могло бы показаться, что пустой вектор бесполезен. Однако, как будет продемонстрировано вскоре, элементы в вектор можно без проблем добавлять и во время выполнения. В действительности наиболее распространенный способ использования векторов подразумевает определение первоначально пустого вектора, в который элементы добавляются по мере необходимости во время выполнения.

Таблица 3.4. Способы инициализации векторов

<code>vector<T> v1</code>	Вектор, содержащий объекты типа <code>T</code> . Стандартный конструктор <code>v1</code> пуст
<code>vector<T> v2(v1)</code>	Вектор <code>v2</code> — копия всех элементов вектора <code>v1</code>
<code>vector<T> v2 = v1</code>	Эквивалент <code>v2(v1)</code> , <code>v2</code> — копия элементов вектора <code>v1</code>
<code>vector<T> v3(n, val)</code>	Вектор <code>v3</code> содержит <code>n</code> элементов со значением <code>val</code>
<code>vector<T> v4(n)</code>	Вектор <code>v4</code> содержит <code>n</code> экземпляров объекта типа <code>T</code> , инициализированного значением по умолчанию
<code>vector<T> v5{ a, b, c ... }</code>	Вектор <code>v5</code> содержит столько элементов, сколько предоставлено инициализаторов; элементы инициализируются соответствующими инициализаторами
<code>vector<T> v5 = { a, b, c ... }</code>	Эквивалент <code>v5{ a, b, c ... }</code>

При определении вектора для его элементов можно также предоставить исходное значение (или значения). Например, можно скопировать элементы из другого вектора. При копировании векторов каждый элемент нового вектора будет копией соответствующего элемента исходного. Оба

вектора должны иметь тот же тип:

```
vector<int> ivect; // первоначально пустой
// присвоить ivect несколько значений
vector<int> ivect2(ivect); // копировать элементы
ivect в ivect2
vector<int> ivect3 = ivect; // копировать элементы
ivect в ivect3
vector<string> svec(ivect2); // svec содержит
строки,
// а не целые числа
```

Списочная инициализация вектора



Согласно новому стандарту, еще одним способом предоставления значений элементам вектора является списочная инициализация (см. раздел 2.2.1), т.е. заключенный в фигурные скобки список любого количества начальных значений элементов:

```
vector<string> articles = {"a", "an", "the"};
```

В результате у вектора будет три элемента: первый со значением "a", второй — "an", последний — "the".

Как уже упоминалось, язык C++ предоставляет несколько форм инициализации (см. раздел 2.2.1). Во многих, но не во всех случаях эти формы инициализации можно использовать взаимозаменяя. На настоящий момент приводились примеры двух форм инициализации: инициализация копией (с использованием знака =) (см. раздел 3.2.1), когда предоставляется только один инициализатор; и внутриклассовая инициализация (см. раздел 2.6.1). Третий способ подразумевает предоставление списка значений элементов, заключенных в фигурные скобки (списочная инициализация). Нельзя предоставить список инициализаторов, используя круглые скобки.

```
vector<string> v1{"a", "an", "the"}; // списочная
инициализация
```

```
vector<string> v2("a", "an", "the"); // ошибка
```

Создание определенного количества элементов

Вектор можно также инициализировать набором из определенного количества элементов, обладающих указанным значением. Счетчик задает

количество элементов, а за ним следует исходное значение для каждого из этих элементов.

```
vector<int> ivect(10, -1);           // десять элементов
типа int, каждый из
                                         // которых
инициализирован значением -1
vector<string> svec(10, "hi!");    // десять строк,
инициализированных
                                         // значением "hi!"
```

Инициализация значения

Иногда инициализирующее значение можно пропустить и указать только размер. В этом случае произойдет *инициализация значения* (value initialization), т.е. библиотека создаст инициализатор элемента сама. Это созданное библиотекой значение используется для инициализации каждого элемента в контейнере. Значение инициализатора элемента вектора зависит от типа его элементов.

Если вектор хранит элементы встроенного типа, такие как `int`, то инициализатором элемента будет значение 0. Если элементы имеют тип класса, такой как `string`, то инициализатором элемента будет его значение по умолчанию.

```
vector<int> ivect(10);           // десять элементов,
инициализированных
                                         // значением 0
vector<string> svec(10);        // десять элементов,
инициализированных
                                         // пустой строкой
```

Эта форма инициализации имеет два ограничения. Первое — некоторые классы всегда требуют явного предоставления инициализатора (см. раздел 2.2.1). Если вектор содержит объекты, тип которых не имеет значения по умолчанию, то начальное значение элемента следует предоставить самому; невозможно создать векторы таких типов, предоставив только размер.

Второе ограничение заключается в том, что при предоставлении количества элементов без исходного значения необходимо использовать *прямую инициализацию* (direct initialization):

```
vector<int> vi = 10; // ошибка: необходима прямая
инициализация
```

Здесь число 10 используется для указания на то, как создать вектор, —

необходимо, чтобы он обладал десятью элементами с инициализированными значениями. Число 10 не "копируется" в вектор. Следовательно, нельзя использовать форму инициализации копией. Более подробная информация об этом ограничении приведена в разделе 7.5.4.



Списочный инициализатор или количество элементов

В некоторых случаях смысл инициализации зависит от того, используются ли при передаче инициализаторов фигурные скобки или круглые. Например, при инициализации вектора `vector<int>` одиночным целочисленным значением это значение могло бы означать либо размер вектора, либо значение элемента. Точно так же, если предоставить два целочисленных значения, то они могли бы быть размером и исходным значением или значениями для двух элементов вектора. Для определения предназначения используются фигурные или круглые скобки.

```
vector<int> v1(10);      // v1 имеет десять элементов
со значением 0
vector<int> v2{10};      // v2 имеет один элемент со
значением 10
vector<int> v3(10, 1);   // v3 имеет десять элементов
со значением 1
vector<int> v4{10, 1};   // v4 имеет два элемента со
значениями 10 и 1
```

Круглые скобки позволяют сообщить, что предоставленные значения должны использоваться для создания объекта. Таким образом, векторы `v1` и `v3` используют свои инициализаторы для определения размера вектора, а также размера и значения его элементов соответственно.

Использование фигурных скобок, `{ . . . }`, означает попытку списочной инициализации. Таким образом, если класс способен использовать значения в фигурных скобках как список инициализаторов элементов, то он так и сделает. Если это невозможно, то следует рассмотреть другие способы инициализации объектов. Значения, предоставленные при инициализации векторов `v2` и `v4`, рассматриваются как значения элементов. Это списочная инициализация объектов; у полученных векторов будет один и два элемента соответственно.

С другой стороны, если используются фигурные скобки и нет никакой

возможности использовать инициализаторы для списочной инициализации объектов, то эти значения будут использоваться для создания объектов. Например, для списочной инициализации вектора строк следует предоставлять значения, которые можно рассматривать как строки. В данном случае нет никаких сомнений, осуществляется ли списочная инициализация элементов или создание вектора указанного размера.

```
vector<string>      v5{ "hi" };           // списочная
инициализация: v5 имеет
                                         // один элемент
vector<string> v6( "hi" ); // ошибка: нельзя создать
вектор из
                                         // строкового литерала
vector<string> v7{ 10 }; // v7 имеет десять
элементов, инициализированных
                                         // значением по умолчанию
vector<string> v8{ 10, "hi" }; // v8 имеет десять
элементов со
                                         // значением "hi"
```

Хотя фигурные скобки использованы во всех этих определениях, кроме одного, только вектор v5 имеет списочную инициализацию. Для списочной инициализации вектора значения в фигурных скобках должны соответствовать типу элемента. Нельзя использовать объект типа int для инициализации строки, поэтому инициализаторы векторов v1 и v8 не могут быть инициализаторами элементов. Если списочная инициализация невозможна, компилятор ищет другие способы инициализации объектов.

Упражнения раздела 3.3.1

Упражнение 3.12. Есть ли ошибки в следующих определениях векторов?

Объясните, что делают допустимые определения. Объясните, почему некорректны недопустимые определения.

- (a) vector<vector<int>> ivec;
- (b) vector<string> svec = ivec;
- (c) vector<string> svec(10, "null");

Упражнение 3.13. Сколько элементов находится в каждом из следующих векторов? Каковы значения этих элементов?

- (a) vector<int> v1; (b) vector<int> v2
(10);
- (c) vector<int> v3(10, 42); (d) vector<int> v4{ 10 };

```
( e)    vector<int> v5{ 10,    42};   ( f)    vector<string>
v6{ 10} ;
( g)    vector<string> v7{ 10, "hi"} ;
```



3.3.2. Добавление элементов в вектор

Прямая инициализация элементов вектора осуществима только при небольшом количестве исходных значений, при копировании другого вектора и при инициализации всех элементов тем же значением. Но обычно при создании вектора неизвестно ни количество его элементов, ни их значения. Но даже если все значения известны, то определение большого количества разных начальных значений может оказаться очень громоздким, чтобы располагать его в месте создания вектора.

Если необходим вектор со значениями от 0 до 9, то можно легко использовать списочную инициализацию. Но что если необходимы элементы от 0 до 99 или от 0 до 999? Списочная инициализация была бы слишком громоздкой. В таких случаях лучше создать пустой вектор и использовать его функцию-член `push_back()`, чтобы добавить элементы во время выполнения. Функция `push_back()` вставляет переданное ей значение в вектор как новый последний элемент. Рассмотрим пример.

```
vector<int> v2; // пустой вектор
for (int i = 0; i != 100; ++i)
    v2.push_back(i); // добавить последовательность
целых чисел в v2
// по завершении цикла v2 имеет 100 элементов со
значениями от 0 до 99
```

Хотя заранее известно, что будет 100 элементов, вектор `v2` определяется как пустой. Каждая итерация добавляет следующее по порядку целое число в вектор `v2` как новый элемент.

Тот же подход используется, если необходимо создать вектор, количество элементов которого до времени выполнения неизвестно. Например, в вектор можно читать введенные пользователем значения.

```
// читать слова со стандартного устройства ввода и
сохранять их
// в векторе как элементы
string word;
```

```
vector<string> text; // пустой вектор
while (cin >> word) {
    text.push_back(word); // добавить слово в текст
}
```

И снова все начинается с пустого вектора. На сей раз, неизвестное количество значений читается и сохраняется в векторе строк `text`.

Ключевая концепция. Рост вектора эффективен

Стандарт требует, чтобы реализация шаблона `vector` обеспечивала эффективное добавление элементов во время выполнения. Поскольку рост вектора эффективен, определение вектора сразу необходимого размера зачастую является ненужным и может даже привести к потере производительности. Исключением является случай, когда все элементы нуждаются в одинаковом значении. При разных значениях элементов обычно эффективней определить пустой вектор и добавлять элементы во время выполнения, по мере того, как значения становятся известны. Кроме того, как будет продемонстрировано в разделе 9.4, шаблон `vector` предоставляет возможности для дальнейшего увеличения производительности при добавлении элементов во время выполнения.

Начало с пустого вектора и добавление элементов во время выполнения кардинально отличается от использования встроенных массивов в языке С и других языках. В частности, если вы знакомы с языком С или Java, то, вероятно, полагаете, что лучше определить вектор в его ожидаемом размере, но фактически имеет место обратное.

Последствия возможности добавления элементов в вектор

Тот факт, что добавление элементов в вектор весьма эффективно, существенно упрощает многие задачи программирования. Но эта простота налагает новые обязательства на наши программы: необходимо гарантировать корректность всех циклов, даже если цикл изменяет размер вектора.

Другое последствие динамического характера векторов станет яснее, когда мы узнаем больше об их использовании. Но есть одно последствие, на которое стоит обратить внимание уже сейчас: по причинам, изложенным в разделе 5.4.3, нельзя использовать серийный оператор `for`, если тело цикла добавляет элементы в вектор.



Тело серийного оператора `f or` не должно изменять размер перебираемой последовательности.

Упражнения раздела 3.3.2

Упражнение 3.14. Напишите программу, читающую последовательность целых чисел из потока `cin` и сохраняющую их в векторе.

Упражнение 3.15. Повторите предыдущую программу, но на сей раз читайте строки.



3.3.3. Другие операции с векторами

Кроме функции `push_back()`, шаблон `vector` предоставляет еще несколько операций, большинство из которых подобно соответствующим операциям класса `string`. Наиболее важные из них приведены в табл. 3.5.

Таблица 3.5. Операции с векторами

<code>v.empty()</code>	Возвращает значение <code>true</code> , если вектор <code>v</code> пуст. В противном случае возвращает значение <code>false</code>
<code>v.size()</code>	Возвращает количество элементов вектора <code>v</code>
<code>v.push_back(t)</code>	Добавляет элемент со значением <code>t</code> в конец вектора <code>v</code>
<code>v[n]</code>	Возвращает ссылку на элемент в позиции <code>n</code> вектора <code>v</code>
<code>v1 = v2</code>	Заменяет элементы вектора <code>v1</code> копией элементов вектора <code>v2</code>
<code>v1 = { a, b, c ... }</code>	Заменяет элементы вектора <code>v1</code> копией элементов из разделяемого запятыми списка
<code>v1 == v2</code> <code>v1 != v2</code>	Векторы <code>v1</code> и <code>v2</code> равны, если они содержат одинаковые элементы в тех же позициях
<code><, <=, >, >=</code>	Имеют обычное значение и полагаются на алфавитный порядок

Доступ к элементам вектора осуществляется таким же способом, как и к символам строки: по их позиции в векторе. Например, для обработки всех элементов вектора можно использовать серийный оператор `for` (раздел 3.2.3).

```
vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9};  
for (auto &i : v) // для каждого элемента вектора v  
    // ( обратите внимание: i - ссылка)  
    i *= i; // квадрат значения элемента  
    for (auto i : v) // для каждого элемента вектора v  
        cout << i << " "; // вывод элемента  
    cout << endl;
```

В первом цикле управляющая переменная `i` определяется как ссылка, чтобы использовать ее для присвоения новых значений элементам вектора

v. Используя спецификатор `auto`, позволим вывести ее тип автоматически. Этот цикл использует новую форму составного оператора присвоения (раздел 1.4.1). Как известно, оператор `+=` добавляет правый операнд к левому и сохраняет результат в левом операнде. Оператор `*=` ведет себя точно так же, но перемножает левый и правый операнды, сохраняя результат в левом операнде. Второй серийный оператор `for` отображает каждый элемент.

Функции-члены `empty()` и `size()` вектора ведут себя так же, как и соответствующие функции класса `string` (раздел 3.2.2): функция `empty()` возвращает логическое значение, указывающее, содержит ли вектор какие-нибудь элементы, а функция `size()` возвращает их количество. Функция-член `size()` возвращает значение типа `size_type`, определенное соответствующим типом шаблона `vector`.



Чтобы использовать тип `size_type`, необходимо указать тип, для которого он определен. Для типа `vector` всегда необходимо указывать тип хранимого элемента (раздел 3.3).

```
vector<int>::size_type // ok
vector::size_type      // ошибка
```

Операторы равенства и сравнения вектора ведут себя как соответствующие операторы класса `string` (раздел 3.2.2). Два вектора равны, если у них одинаковое количество элементов и значения соответствующих элементов совпадают. Операторы сравнения полагаются на алфавитный порядок: если у векторов разные размеры, но соответствующие элементы равны, то вектор с меньшим количеством элементов меньше вектора с большим количеством элементов. Если у элементов векторов разные значения, то их отношения определяются по первым отличающимся элементам.

Сравнить два вектора можно только в том случае, если возможно сравнить элементы этих векторов. Некоторые классы, такие как `string`, определяют смысл операторов равенства и сравнения. Другие, такие как класс `Sales_item`, этого не делают. Операции, поддерживаемые классом `Sales_item`, перечислены в разделе 1.5.1. Они не включают ни операторов равенства, ни сравнения. В результате нельзя сравнить два

вектора объектов класса `Sales_item`.

Вычисление индекса вектора

Используя оператор индексирования (раздел 3.2.3), можно выбрать указанный элемент. Подобно строкам, индексирование вектора начинаются с 0; индекс имеет тип `size_type` соответствующего типа; и если вектор не константен, то в возвращенный оператором индексирования элемент можно осуществить запись. Кроме того, как было продемонстрировано в разделе 3.2.3, можно вычислить индекс и непосредственно обратиться к элементу в данной позиции.

Предположим, имеется набор оценок степеней от 0 до 100. Необходимо рассчитать, сколько оценок попадает в кластер по 10. Между нулем и 100 возможна 101 оценка. Эти оценки могут быть представлены 11 кластерами: 10 кластеров по 10 оценок каждый плюс один кластер для наивысшей оценки 100. Первый кластер подсчитывает оценки от 0 до 9, второй — от 10 до 19 и т.д. Заключительный кластер подсчитывает количество оценок 100.

Таким образом, если введены следующие оценки:

42 65 95 100 39 67 95 76 88 76 83 92 76 93

результат их кластеризации должен быть таким:

0 0 0 1 1 0 2 3 2 4 1

Он означает, что не было никаких оценок ниже 30, одна оценка в 30-х, одна в 40-х, ни одной в 50-х, две в 60-х, три в 70-х, две в 80-х, четыре в 90-х и одна оценка 100.

Используем для содержания счетчиков каждого кластера вектор с 11 элементами. Индекс кластера для данной оценки можно определить делением этой оценки на 10. При делении двух целых чисел получается целое число, дробная часть которого усекается. Например, $42/10=4$, $65/10=6$, а $100/10=10$.

Как только индекс кластера будет вычислен, его можно использовать для индексирования вектора и доступа к счетчику, значение которого необходимо увеличить.

```
// подсчет количества оценок в кластере по десять:  
0--9,  
// 10--19, ... 90--99, 100  
vector<unsigned> scores(11, 0); // 11 ячеек, все со  
значением 0  
unsigned grade;  
while (cin >> grade) { // читать оценки
```

```
if (grade <= 100) // обрабатывать только  
допустимые оценки  
    ++scores[grade/10]; // приращение счетчика  
текущего кластера
```

Код начинается с определения вектора для хранения счетчиков кластеров. В данном случае все элементы должны иметь одинаковое значение, поэтому резервируем 11 элементов, каждый из которых инициализируем значением 0. Условие цикла `while` читает оценки. В цикле проверяется допустимость значения прочитанной оценки (т.е. оно меньше или равно 100). Если оценка допустима, то увеличиваем соответствующий счетчик.

Оператор, осуществляющий приращение, является хорошим примером краткости кода C++:

```
++scores[grade/10]; // приращение счетчика текущего  
кластера
```

Это выражение эквивалентно следующему:

```
auto ind = grade/10; // получить индекс  
ячейки  
scores[ind] = scores[ind] + 1; // приращение  
счетчика
```

Индекс ячейки вычисляется делением значения переменной `grade` на 10. Полученный результат используется для индексирования вектора `scores`, что обеспечивает доступ к соответствующему счетчику для этой оценки. Увеличение значения этого элемента означает принадлежность текущей оценки данному диапазону.

Как уже упоминалось, при использовании индексирования следует позаботиться о том, чтобы индексы оставались в диапазоне допустимых значений (см. раздел 3.2.3). В этой программе проверка допустимости подразумевает принадлежность оценки к диапазону 0–100. Таким образом, можно использовать индексы от 0 до 10. Они расположены в пределах от 0 до `scores.size() - 1`.

Индексация не добавляет элементов

Новички в C++ иногда полагают, что индексирование вектора позволяет добавлять в него элементы, но это не так. Следующий код намеревается добавить десять элементов в вектор `i vec`:

```
vector<int> i vec; // пустой вектор  
for (decltype(i vec.size()) ix = 0; ix != 10; ++ix)
```

```
ivec[ ix] = ix; // катастрофа: ivec не имеет  
элементов
```

Причина ошибки — вектор `ivec` пуст; в нем нет никаких элементов для индексирования! Как уже упоминалось, правильный цикл использовал бы функцию `push_back()`:

```
for (decltype(ivec.size()) ix = 0; ix != 10; ++ix)  
    ivec.push_back(ix); // ok: добавляет новый элемент  
со значением ix
```



Оператор индексирования вектора (и строки) лишь выбирает существующий элемент; он *не может* добавить новый элемент.

Внимание! Индексировать можно лишь существующие элементы!

Очень важно понять, что оператор индексирования (`[]`) можно использовать для доступа только к фактически существующим элементам. Рассмотрим пример.

```
vector<int> ivec; // пустой вектор  
cout << ivec[0]; // ошибка: ivec не имеет  
элементов!
```

```
vector<int> ivec2(10); // вектор из 10 элементов  
cout << ivec2[10]; // ошибка: ivec2 имеет  
элементы 0...9
```

Попытка обращения к несуществующему элементу является серьезной ошибкой, которую вряд ли обнаружит компилятор. В результате будет получено случайное значение.

Попытка индексирования несуществующих элементов, к сожалению, является весьма распространенной и грубой ошибкой программирования. Так называемая ошибка *переполнения буфера* (buffer overflow) — результат индексирования несуществующих элементов. Такие ошибки являются наиболее распространенной причиной проблем защиты приложений.



Наилучший способ гарантировать невыход индекса из диапазона — это избежать индексации вообще. Для этого везде, где только возможно, следует использовать серийный оператор `for`.

Упражнения раздела 3.3.3

Упражнение 3.16. Напишите программу, выводящую размер и содержимое вектора из упражнения 3.13. Проверьте правильность своих ответов на это упражнение. При неправильных ответах повторно изучите раздел 3.3.1.

Упражнение 3.17. Прочитайте последовательность слов из потока `cin` и сохраните их в векторе. Прочитав все слова, обработайте вектор и переведите символы каждого слова в верхний регистр. Отобразите преобразованные элементы по восемь слов на строку.

Упражнение 3.18. Корректна ли следующая программа? Если нет, то как ее исправить?

```
vector<int> iVec;
iVec[ 0 ] = 42;
```

Упражнение 3.19. Укажите три способа определения вектора и заполнения его десятью элементами со значением 42. Укажите, есть ли предпочтительный способ для этого и почему.

Упражнение 3.20. Прочитайте набор целых чисел в вектор. Отобразите сумму каждой пары соседних элементов. Измените программу так, чтобы она отображала сумму первого и последнего элементов, затем сумму второго и предпоследнего и т.д.



3.4. Знакомство с итераторами

Хотя для доступа к символам строки или элементам вектора можно использовать индексирование, для этого существует и более общий механизм — *итераторы* (iterator). Как будет продемонстрировано в части II, кроме векторов библиотека предоставляет несколько других видов контейнеров. У всех библиотечных контейнеров есть итераторы, но только некоторые из них поддерживают оператор индексирования. С технической точки зрения тип `string` не является контейнерным, но он поддерживает большинство контейнерных операций. Как уже упоминалось, и строки, и векторы предоставляют оператор индексирования. У них также есть итераторы.

Как и указатели (см. раздел 2.3.2), итераторы обеспечивают косвенный доступ к объекту. В случае итератора этим объектом является элемент в контейнере или символ в строке. Итератор позволяет выбрать элемент, а также поддерживает операции перемещения с одного элемента на другой. Подобно указателям, итератор может быть допустим или недопустим. Допустимый итератор указывает либо на элемент, либо на позицию за последним элементом в контейнере. Все другие значения итератора недопустимы.



3.4.1. Использование итераторов

В отличие от указателей, для получения итератора не нужно использовать оператор обращения к адресу. Для этого обладающие итераторами типы имеют члены, возвращающие эти итераторы. В частности, они обладают функциями-членами `begin()` и `end()`. Функция-член `begin()` возвращает итератор, который обозначает первый элемент (или первый символ), если он есть.

```
// типы b и e определяют компилятор; см. раздел  
2.5.2
```

```
// b обозначает первый элемент контейнера v, а e -  
элемент
```

```
// после последнего
```

```
auto b = v.begin(), e = v.end();
```

```
// b и e имеют одинаковый тип
```

Итератор, возвращенный функцией `end()`, указывает на следующую позицию за концом контейнера (или строки). Этот итератор обозначает несуществующий элемент за концом контейнера. Он используется как индикатор, означающий, что обработаны все элементы. Итератор, возвращенный функцией `end()`, называют *итератором после конца* (off-the-end iterator), или сокращенно *итератором end*. Если контейнер пуст, функция `begin()` возвращает тот же итератор, что и функция `end()`.



Если контейнер пуст, возвращаемые функциями `begin()` и `end()` итераторы совпадают и, оба являются итератором после конца.

Обычно точный тип, который имеет итератор, неизвестен (да и не нужен). В этом примере при определении итераторов `b` и `e` использовался спецификатор `auto` (см. раздел 2.5.2). В результате тип этих переменных будет совпадать с возвращаемыми функциями-членами `begin()` и `end()` соответственно. Не будем пока распространяться об этих типах.

Операции с итераторами

Итераторы поддерживают лишь несколько операций, которые перечислены в табл. 3.6. Два допустимых итератора можно сравнить при помощи операторов `==` и `!=`. Итераторы равны, если они указывают на тот же элемент или если оба они указывают на позицию после конца того же контейнера. В противном случае они не равны.

Таблица 3.6. Стандартные операции с итераторами контейнера

<code>*iter</code>	Возвращает ссылку на элемент, обозначенный итератором <code>iter</code>
<code>iter->mem</code>	Обращение к значению итератора <code>iter</code> и выборка члена <code>mem</code> основного элемента. Эквивалент <code>(*iter).mem</code>
<code>++iter</code>	Инкремент итератора <code>iter</code> для обращения к следующему элементу контейнера
<code>--iter</code>	Декремент итератора <code>iter</code> для обращения к предыдущему элементу контейнера
<code>iter1 == iter2</code> <code>iter1 != iter2</code>	Сравнивает два итератора на равенство (неравенство). Два итератора равны, если они указывают на тот же элемент или на следующий элемент после конца того же контейнера

Подобно указателям, к значению итератора можно обратиться, чтобы получить элемент, на который он ссылается. Кроме того, подобно указателям, можно обратиться к значению только допустимого итератора, который обозначает некий элемент (см. раздел 2.3.2). Результат обращения к значению недопустимого итератора или итератора после конца непредсказуем.

Перепишем программу из раздела 3.2.3, преобразующую строчные символы строки в прописные, с использованием итератора вместо индексирования:

```
string s("some string");
if (s.begin() != s.end()) { // удостовериться, что
строка s не пуста
    auto it = s.begin();           // it указывает на
первый символ строки s
    *it = toupper(*it);          // текущий символ в
верхний регистр
}
```

Как и в первоначальной программе, сначала удостоверимся, что строка `s` не пуста. В данном случае для этого сравниваются итераторы, возвращенные функциями `begin()` и `end()`. Эти итераторы равны, если строка пуста. Если они не равны, то в строке `s` есть по крайней мере один

символ.

В теле оператора `if` функция `begin()` возвращает итератор на первый символ, который присваивается переменной `it`. Обращение к значению этого итератора и передача его функции `toupper()` позволяет перевести данный символ в верхний регистр. Кроме того, обращение к значению итератора `it` слева от оператора присвоения позволяет присвоить символ, возвращенный функцией `toupper()`, первому символу строки `s`. Как и в первоначальной программе, вывод будет таким:

Some string

Перемещение итератора с одного элемента на другой

Итераторы используют оператор инкремента (оператор `++`) (см. раздел 1.4.1) для перемещения с одного элемента на следующий. Операция приращения итератора логически подобна приращению целого числа. В случае целых чисел результатом будет целочисленное значение на единицу больше 1. В случае итераторов результатом будет перемещение итератора на одну позицию.



Поскольку итератор, возвращенный функцией `end()`, не указывает на элемент, он не допускает ни приращения, ни обращения к значению.

Перепишем программу, изменяющую регистр первого слова в строке, с использованием итератора.

```
// обрабатывать символы, пока они не исчерпаются,
// или не встретится пробел
for (auto it = s.begin(); it != s.end() &&
!isspace(*it); ++it)
    *it = toupper(*it); // преобразовать в верхний
регистр
```

Этот цикл, подобно таковому в разделе 3.2.3, перебирает символы строки `s`, останавливаясь, когда встречается пробел. Но данный цикл использует для этого итератор, а не индексирование.

Цикл начинается с инициализации итератора `it` результатом вызова функции `s.begin()`, чтобы он указывал на первый символ строки `s` (если он есть). Условие проверяет, не достиг ли итератор `it` конца строки

(`s.end()`). Если это не так, то проверяется следующее условие, где обращение к значению итератора `it`, возвращающее текущий символ, передается функции `isspace()`, чтобы выяснить, не пробел ли это. В конце каждой итерации выполняется оператор `++it`, чтобы переместить итератор на следующий символ строки `s`.

У этого цикла то же тело, что и у последнего оператора `if` предыдущей программы. Обращение к значению итератора `it` используется и для передачи текущего символа функции `toupper()`, и для присвоения полученного результата символу, на который указывает итератор `it`.

Ключевая концепция. Обобщенное программирование

Программисты, перешедшие на язык C++ с языка С или Java, могли бы быть удивлены тем, что в данном цикле `for` был использован оператор `!=`, а не `<`. Программисты C++ используют оператор `!=` исключительно по привычке. По этой же причине они используют итераторы, а не индексирование: этот стиль программирования одинаково хорошо применим к контейнерам различных видов, предоставляемых библиотекой.

Как уже упоминалось, только у некоторых библиотечных типов, `vector` и `string`, есть оператор индексирования. Тем не менее у всех библиотечных контейнеров есть итераторы, для которых определены операторы `==` и `!=`. Однако большинство их итераторов не имеют оператора `<`. При обычном использовании итераторов и оператора `!=` можно не заботиться о точном типе обрабатываемого контейнера.

Типы итераторов

Подобно тому, как не всегда известен точный тип `size_type` элемента вектора или строки (см. раздел 3.2.2), мы обычно не знаем (да и не обязаны знать) точный тип итератора. Как и в случае с типом `size_type`, библиотечные типы, у которых есть итераторы, определяют типы по имени `iterator` и `const_iterator`, которые представляют фактические типы итераторов.

```
vector<int>::iterator it;           // it позволяет
читать и записывать
                                         // в элементы
вектора vector<int>
string::iterator it2;                 // it2 позволяет
читать и записывать
```

```

    // СИМВОЛЫ в
строку
vector<int>::const_iterator it3; // it3 позволяет
читать, но не
// записывать

Элементы
string::const_iterator it4; // it4 позволяет
читать, но не
// записывать

СИМВОЛЫ

```

Тип `const_iterator` ведет себя как константный указатель (см. раздел 2.4.2). Как и константный указатель, тип `const_iterator` позволяет читать, но не писать в элемент, на который он указывает; объект типа `iterator` позволяет и читать, и записывать. Если вектор или строка являются константой, можно использовать итератор только типа `const_iterator`. Если вектор или строка на являются константой, можно использовать итератор и типа `iterator`, и типа `const_iterator`.

Терминология. Итераторы и типы итераторов

Термин *итератор* (`iterator`) используется для трех разных сущностей. Речь могла бы идти о *концепции* итератора, или о *типе* `iterator`, определенном классом контейнера, или об *объекте* итератора.

Следует уяснить, что существует целый набор типов, связанных концептуально. Тип относится к итераторам, если он поддерживает общепринятый набор функций. Эти функции позволяют обращаться к элементу в контейнере и переходить с одного элемента на другой.

Каждый класс контейнера определяет тип по имени `iterator`, который обеспечивает действия концептуального итератора.

Функции `begin()` и `end()`

Тип, возвращаемый функциями `begin()` и `end()`, зависит от константности объекта, для которого они были вызваны. Если объект является константой, то функции `begin()` и `end()` возвращают итератор типа `const_iterator`; если объект не константа, они возвращают итератор типа `iterator`.

```

vector<int> v;
const vector<int> cv;

```

```
auto it1 = v.begin(); // it1 имеет тип  
vector<int>::iterator  
auto it2 = cv.begin(); // it2 имеет тип  
vector<int>::const_iterator
```



Зачастую это стандартное поведение желательно изменить. По причинам, рассматриваемым в разделе 6.2.3, обычно лучше использовать константный тип (такой как `const_iterator`), когда необходимо только читать, но не записывать в объект. Чтобы позволить специально задать тип `const_iterator`, новый стандарт вводит две новые функции, `cbegin()` и `cend()`:

```
auto it3 = v.cbegin(); // it3 имеет тип  
vector<int>::const_iterator
```

Подобно функциям-членам `begin()` и `end()`, эти функции-члены возвращают итераторы на первый и следующий после последнего элементы контейнера. Но независимо от того, является ли вектор (или строка) константой, они возвращают итератор типа `const_iterator`.

Объединение обращения к значению и доступа к члену

При обращении к значению итератора получается объект, на который указывает итератор. Если этот объект имеет тип класса, то может понадобиться доступ к члену полученного объекта. Например, если есть вектор строк, то может понадобиться узнать, не пуст ли некий элемент. С учетом, что `it` — это итератор данного вектора, можно следующим образом проверить, не пуста ли строка, на которую он указывает:

```
(*it).empty()
```

По причинам, рассматриваемым в разделе 4.1.2, круглые скобки в части `(*it).empty()` необходимы. Круглые скобки требуют применить оператор обращения к значению к итератору `it`, а к результату применить точечный оператор (см. раздел 1.5.2). Без круглых скобок точечный оператор относился бы к итератору `it`, а не к полученному объекту.

```
(*it).empty() // обращение к значению it и вызов  
функции-члена empty()  
// полученного объекта  
*it.empty() // ошибка: попытка вызова функции-члена  
empty()  
// итератора it,
```

```
// но итератор it не имеет функции-  
члена empty()
```

Второе выражение интерпретируется как запрос на выполнение функции-члена `empty()` объекта `it`. Но `it` — это итератор, и он не имеет такой функции. Следовательно, второе выражение ошибочно.

Чтобы упростить такие выражения, язык предоставляет *оператор стрелки* (arrow operator) (оператор `->`). Оператор стрелки объединяет обращение к значению и доступ к члену. Таким образом, выражение `it->mem` является синоним выражения `(*it).mem`.

Предположим, например, что имеется вектор `vector<string>` по имени `text`, содержащий данные из текстового файла. Каждый элемент вектора — это либо предложение, либо пустая строка, представляющая конец абзаца. Если необходимо отобразить содержимое первого параграфа из вектора `text`, то можно было бы написать цикл, который перебирает вектор `text`, пока не встретится пустой элемент.

```
// отобразить каждую строку вектора text до первой  
пустой строки
```

```
for (auto it = text.cbegin();  
     it != text.cend() && !it->empty(); ++it)  
    cout << *it << endl;
```

Код начинается с инициализации итератора `it` указанием на первый элемент вектора `text`. Цикл продолжается до тех пор, пока не будут обработаны все элементы вектора `text` или пока не встретится пустой элемент. Пока есть элементы и текущий элемент не пуст, он отображается. Следует заметить, что, поскольку цикл только читает элементы, но не записывает их, здесь для управления итерацией используются функции `cbegin()` и `cend()`.

Некоторые операции с векторами делают итераторы недопустимыми

В разделе 3.3.2 упоминался тот факт, что векторы способны расти динамически. Обращалось также внимание на то, что нельзя добавлять элементы в вектор в цикле серийного оператора `for`. Еще одно замечание: любая операция, такая как вызов функции `push_back()`, изменяет размер вектора и способна сделать недопустимыми все итераторы данного вектора. Более подробная информация по этой теме приведена в разделе 9.3.6.



На настоящий момент достаточно знать, что использующие итераторы цикла не должны добавлять элементы в контейнер, с которым связаны итераторы.



3.4.2. Арифметические действия с итераторами

Инкремент итератора перемещает его на один элемент. Инкремент поддерживают итераторы всех библиотечных контейнеров. Аналогично операторы `==` и `!=` можно использовать для сравнения двух допустимых итераторов (см. раздел 3.4) любых библиотечных контейнеров.

Итераторы строк и векторов поддерживают дополнительные операции, позволяющие перемещать итераторы на несколько позиций за раз. Они также поддерживают все операторы сравнения. Эти операторы зачастую называют *арифметическими действиями с итераторами* (iterator arithmetic). Они приведены в табл. 3.7.

Таблица 3.7. Операции с итераторами векторов и строк

<code>iter + n</code> <code>iter - n</code>	Добавление (вычитание) целочисленного значения <code>n</code> к (из) итератору возвращает итератор, указывающий на элемент <code>n</code> позиций вперед (назад) в пределах контейнера. Полученный итератор должен указывать на элемент или на следующую позицию за концом того же контейнера
<code>iter1 += n</code> <code>iter1 -= n</code>	Составные операторы присвоения со сложением и вычитанием итератора. Присваивает итератору <code>iter1</code> значение на <code>n</code> позиций больше или меньше предыдущего
<code>iter1 - iter2</code>	Вычитание двух итераторов возвращает значение, которое, будучи добавлено к правому итератору, вернет левый. Итераторы должны указывать на элементы или на следующую позицию за концом того же контейнера
<code>>, >=, <, <=</code>	Операторы сравнения итераторов. Один итератор меньше другого, если он указывает на элемент, расположенный в контейнере ближе к началу. Итераторы должны указывать на элементы или на следующую позицию за концом того же контейнера

Арифметические операции с итераторами

К итератору можно добавить (или вычесть из него) целочисленное

значение. Это вернет итератор, перемещенный на соответствующее количество позиций вперед (или назад). При добавлении или вычитании целочисленного значения из итератора результат должен указывать на элемент в том же векторе (или строке) или на следующую позицию за концом того же вектора (или строки). В качестве примера вычислим итератор на элемент, ближайший к середине вектора:

```
// вычислить итератор на элемент, ближайший к  
середине вектора vi
```

```
auto mid = vi.begin() + vi.size() / 2;
```

Если у вектора `vi` 20 элементов, то результатом `vi.size() / 2` будет 10. В данном случае переменной `mid` будет присвоено значение, равное `vi.begin() + 10`. С учетом, что нумерация индексов начинается с 0, это тот же элемент, что и `vi[10]`, т.е. элемент на десять позиций от начала.

Кроме сравнения двух итераторов на равенство, итераторы векторов и строк можно сравнить при помощи операторов сравнения (`<`, `<=`, `>`, `>=`). Итераторы должны быть допустимы, т.е. должны обозначать элементы (или следующую позицию за концом) того же вектора или строки. Предположим, например, что `it` является итератором в том же векторе, что и `mid`. Следующим образом можно проверить, указывает ли итератор `it` на элемент до или после итератора `mid`:

```
if (it < mid)  
    // обработать элементы в первой половине вектора  
vi
```

Можно также вычесть два итератора, если они указывают на элементы (или следующую позицию за концом) того же вектора или строки. Результат — дистанция между итераторами. Под дистанцией подразумевается значение, на которое следует изменить один итератор, чтобы получить другой. Результат имеет целочисленный знаковый тип `difference_type`. Тип `difference_type` определен и для вектора, и для строки. Этот тип знаковый, поскольку результатом вычитания может оказаться отрицательное значение.

Использование арифметических действий с итераторами

Классическим алгоритмом, использующим арифметические действия с итераторами, является *двоичный поиск* (binary search). Двоичный (бинарный) поиск ищет специфическое значение в отсортированной последовательности. Алгоритм работает так: сначала исследуется элемент,

ближайший к середине последовательности. Если это искомый элемент, работа закончена. В противном случае, если этот элемент меньше искомого, поиск продолжается только среди элементов после исследованного. Если средний элемент больше искомого, поиск продолжается только в первой половине. Вычисляется новый средний элемент оставшегося диапазона, и действия продолжаются, пока искомый элемент не будет найден или пока не исчерпаются элементы.

Используя итераторы, двоичный поиск можно реализовать следующим образом:

```
// текст должен быть отсортирован
// beg и end ограничивают диапазон, в котором
осуществляется поиск
auto beg = text.begin(), end = text.end();
auto mid = text.begin() + (end - beg)/2; // исходная середина
// пока еще есть элементы и искомый не найден
while (mid != end && *mid != sought) {
    if (sought < *mid) // находится ли искомый элемент
        в первой половине?
        end = mid; // если да, то изменить
диапазон, игнорируя вторую
        // половину
    else // искомый элемент во второй
половине
        beg = mid + 1; // начать поиск с элемента
разу после середины
    mid = beg + (end - beg)/2; // новая середина
}
```

Код начинается с определения трех итераторов: `beg` будет первым элементом в диапазоне, `end` — элементом после последнего, а `mid` — ближайшим к середине. Инициализируем эти итераторы значениями, охватывающими весь диапазон вектора `vector<string>` по имени `text`.

Сначала цикл проверяет, не пуст ли диапазон. Если значение итератора `mid` равно текущему значению итератора `end`, то элементы для поиска исчерпаны. В таком случае условие ложно и цикл `while` завершается. В противном случае итератор `mid` указывает на элемент, который проверяется на соответствие искомому. Если это так, то цикл завершается.

Если элементы все еще есть, код в цикле `while` корректирует диапазон, перемещая итератор `end` или `beg`. Если обозначенный итератором `mid` элемент больше, чем `sought`, то если искомый элемент и есть в векторе, он находится перед элементом, обозначенным итератором `mid`. Поэтому можно игнорировать элементы после середины, что мы и делаем, присваивая значение итератора `mid` итератору `end`. Если значение $*mid$ меньше, чем `sought`, элемент должен быть в диапазоне элементов после обозначенного итератором `mid`. В данном случае диапазон корректируется присвоением итератору `beg` позиции сразу после той, на которую указывает итератор `mid`. Уже известно, что `mid` не указывает на искомый элемент, поэтому его можно исключить из диапазона.

В конце цикла `while` итератор `mid` будет равен итератору `end` либо будет указывать на искомый элемент. Если итератор `mid` равен `end`, то искомого элемента нет в векторе `text`.

Упражнения раздела 3.4.2

Упражнение 3.24. Переделайте последнее упражнение раздела 3.3.3 с использованием итераторов.

Упражнение 3.25. Перепишите программу кластеризации оценок из раздела 3.3.3 с использованием итераторов вместо индексации.

Упражнение 3.26. Почему в программе двоичного поиска использован код `mid = beg + (end - beg) / 2;`, а не `mid = (beg + end) / 2; ?`

3.5. Массивы

Массив (`array`) — это структура данных, подобная библиотечному типу `vector` (см. раздел 3.3), но с другим соотношением между производительностью и гибкостью. Как и вектор, массив является контейнером безымянных объектов одинакового типа, к которым обращаются по позиции. В отличие от вектора, массивы имеют фиксированный размер; добавлять элементы к массиву нельзя. Поскольку размеры массивов постоянны, они иногда обеспечивают лучшую производительность во время выполнения приложений. Но это преимущество приобретается за счет потери гибкости.



Если вы не знаете точно, сколько элементов необходимо, используйте вектор.

3.5.1. Определение и инициализация встроенных массивов

Массив является составным типом (см. раздел 2.3). Оператор объявления массива имеет форму `a[d]`, где `a` — имя; `d` — размерность определяемого массива. Размерность задает количество элементов массива, она должна быть больше нуля. Количество элементов — это часть типа массива, поэтому она должна быть известна на момент компиляции. Следовательно, размерность должна быть константным выражением (см. раздел 2.4.4).

```
unsigned cnt = 42; // неконстантное
выражение
constexpr unsigned sz = 42; // константное
выражение
// constexpr см. р.
```

2. 4. 4

```
int arr[10]; // массив десяти целых
чисел
int *parr[ sz ]; // массив 42 указателей
на int
```

```

string bad[ cnt];                                // ошибка: cnt
неконстантное выражение
string strs[ get_size( )];          // ok, если get_size -
constexpr,                                     // в противном случае -
                                            // ошибка

```

По умолчанию элементы массива инициализируются значением по умолчанию (раздел 2.2.1).



Подобно переменным встроенного типа, инициализированный по умолчанию массив встроенного типа, определенный в функции, будет содержать неопределенные значения.

При определении массива необходимо указать тип его элементов. Нельзя использовать спецификатор `auto` для вывода типа из списка инициализаторов. Подобно вектору, массив содержит объекты. Таким образом, невозможен массив ссылок.

Явная инициализация элементов массива

Массив допускает списочную инициализацию (см. раздел 3.3.1) элементов. В этом случае размерность можно опустить. Если размерность отсутствует, компилятор выводит ее из количества инициализаторов. Если размерность определена, количество инициализаторов не должно превышать ее.

Если размерность больше количества инициализаторов, то инициализаторы используются для первых элементов, а остальные инициализируются по умолчанию (см. раздел 3.3.1):

```

const unsigned sz = 3;
int ia1[ sz ] = { 0, 1, 2 };           // массив из трех
целых чисел со                               // значениями 0, 1, 2
                                              // массив размером 3
элемента
int a2[ ] = { 0, 1, 2 };                 // эквивалент a3[ ] =
                                              // { 0, 1, 2, 0, 0 }
int a3[ 5 ] = { 0, 1, 2 };                // эквивалент a4[ ] =
{ 0, 1, 2, 0, 0 }
string a4[ 3 ] = { "hi", "bye" };        // эквивалент a4[ ] =

```

```
{ "hi", "bye", "" }  
    int a5[ 2 ] = { 0, 1, 2 }; // ошибка: слишком  
много инициализаторов
```

Особенности символьных массивов

У символьных массивов есть дополнительная форма инициализации: строковым литералом (см. раздел 2.1.3). Используя эту форму инициализации, следует помнить, что строковые литералы заканчиваются нулевым символом. Этот нулевой символ копируется в массив наряду с символами литерала.

```
char a1[ ] = { 'C', '+', '+' }; // списочная  
инициализация без // нулевого  
символа  
char a2[ ] = { 'C', '+', '+', '\0' }; // списочная  
инициализация с явным // нулевым  
символом  
char a3[ ] = "C++"; // нулевой  
символ добавляется // автоматически  
const char a4[ 6 ] = "Daniel"; // ошибка: нет  
места для нулевого // символа!
```

Массив a1 имеет размерность 3; массивы a2 и a3 — размерности 4. Определение массива a4 ошибочно. Хотя литерал содержит только шесть явных символов, массив a4 должен иметь по крайней мере семь элементов, т.е. шесть для самого литерала и один для нулевого символа.

Не допускается ни копирование, ни присвоение

Нельзя инициализировать массив как копию другого массива, не допустимо также присвоение одного массива другому.

```
int a[ ] = { 0, 1, 2 }; // массив из трех целых чисел  
int a2[ ] = a; // ошибка: нельзя  
инициализировать один массив // другим  
a2 = a; // ошибка: нельзя присваивать  
один массив другому
```



ВНИМАНИЕ

Некоторые компиляторы допускают присвоение массивов при применении *расширения компилятора* (compiler extension). Как правило, использования нестандартных средств следует избегать, поскольку они не будут работать на других компиляторах.

Понятие сложных объявлений массива

Как и векторы, массивы способны содержать объекты большинства типов. Например, может быть массив указателей. Поскольку массив — это объект, можно определять и указатели, и ссылки на массивы. Определение массива, содержащего указатели, довольно просто, определение указателя или ссылки на массив немного сложней.

```
int *ptrs[ 10 ]; // ptrs массив десяти
указателей на int
int &refs[ 10 ] = /* ? */; // ошибка: массив ссылок
невозможен
int ( *Parray )[ 10 ] = &arr; // Parray указывает на
массив из десяти int
int ( &arrRef )[ 10 ] = arr; // arrRef ссылается на
массив из десяти ints
```

Обычно модификаторы типа читают справа налево. Читаем определение `ptrs` справа налево (см. раздел 2.3.3): определить массив размером 10 по имени `ptrs` для хранения указателей на тип `int`.

Определение `Parray` также стоит читать справа налево. Поскольку размерность массива следует за объявляемым именем, объявление массива может быть легче читать изнутри наружу, а не справа налево. Так намного проще понять тип `Parray`. Объявление начинается с круглых скобок вокруг части `*Parray`, означающей, что `Parray` — указатель. Глядя направо, можно заметить, что указатель `Parray` указывает на массив размером 10. Глядя влево, можно заметить, что элементами этого массива являются целые числа. Таким образом, `Parray` — это указатель на массив из десяти целых чисел. Точно так же часть `(&arrRef)` означает, что `arrRef` — это ссылка, а типом, на который она ссылается, является массив размером 10, хранящий элементы типа `int`.

Конечно, нет никаких ограничений на количество применяемых

модификаторов типа.

```
int *(&arry)[10] =ptrs; // arry - ссылка на массив из десяти указателей
```

Читая это объявление изнутри наружу, можно заметить, что `arry` — это ссылка. Глядя направо, можно заметить, что объект, на который ссылается `arry`, является массивом размером 10. Глядя влево, можно заметить, что типом элемента является указатель на тип `int`. Таким образом, `arry` — это ссылка на массив десяти указателей.



Зачастую объявление массива может быть проще понять, начав его чтение с имени массива и продолжив его изнутри наружу.

Упражнения раздела 3.5.1

Упражнение 3.27. Предположим, что функция `txt_size()` на получает никаких аргументов и возвращают значение типа `int`. Объясните, какие из следующих определений недопустимы и почему?

```
unsigned buf_size = 1024;
(a) int ia[buf_size];    (b) int ia[4 * 7 - 14];
(c) int ia[txt_size()];  (d) char st[11] =
"fundamental";
```

Упражнение 3.28. Какие значения содержатся в следующих массивах?

```
string sa[10];
int ia[10];
int main() {
    string sa2[10];
    int ia2[10];
}
```

Упражнение 3.29. Перечислите некоторые из недостатков использования массива вместо вектора.

3.5.2. Доступ к элементам массива

Подобно библиотечным типам `vector` и `string`, для доступа к элементам массива можно использовать серийный оператор `for` или *оператор индексирования* (`[]`) (subscript). Как обычно, индексы начинаются с 0. Для массива из десяти элементов используются индексы от

0 до 9, а не от 1 до 10.

При использовании переменной для индексирования массива ее обычно определяют как имеющую тип `size_t`. Тип `size_t` — это машинозависимый беззнаковый тип, гарантированно достаточно большой для содержания размера любого объекта в памяти. Тип `size_t` определен в заголовке `csstddef`, который является версией C++ заголовка `stddef.h` библиотеки С.

За исключением фиксированного размера, массивы используются подобно векторам. Например, можно повторно реализовать программу оценок из раздела 3.3.3, используя для хранения счетчиков кластеров массив.

```
// подсчет количества оценок в кластере по десять:  
0--9,  
// 10--19, ... 90--99, 100  
unsigned scores[11] = {};// 11 ячеек, все со  
значением 0  
unsigned grade;  
while (cin >> grade) {  
    if (grade <= 100)  
        ++scores[grade/10]; // приращение счетчика  
текущего кластера  
}
```

Единственное очевидное различие между этой программой и приведенной в разделе 3.3.3 в объявлении массива `scores`. В данной программе это массив из 11 элементов типа `unsigned`. Не столь очевидно то различие, что оператор индексирования в данной программе тот, который определен как часть языка. Этот оператор применяется с operandами типа массива. Оператор индексирования, используемый в программе в разделе 3.3.3, был определен библиотечным шаблоном `vector` и применялся к operandам типа `vector`.

Как и в случае строк или векторов, для перебора всего массива лучше использовать серийный оператор `for`. Например, все содержимое массива `scores` можно отобразить следующим образом:

```
for (auto i : scores) // для каждого счетчика в  
scores  
    cout << i << " "; // отобразить его значение  
    cout << endl;
```

Поскольку размерность является частью типа каждого массива, системе

известно количество элементов в массиве `scores`. Используя средства серийного оператора `for`, перебором можно управлять и не самостоятельно.

Проверка значений индекса

Как и в случае со строкой и вектором, ответственность за невыход индекса за пределы массива лежит на самом программисте. Он сам должен гарантировать, что значение индекса будет больше или равно нулю, но не больше размера массива. Ничто не мешает программе перешагнуть границу массива, кроме осторожности и внимания разработчика, а также полной проверки кода. В противном случае программа будет компилироваться и выполняться правильно, но все же содержать скрытую ошибку, способную проявиться в наименее подходящий момент.



Наиболее распространенным источником проблем защиты приложений является ошибка переполнения буфера. Причиной такой ошибки является отсутствие в программе проверки индекса, в результате чего программа ошибочно использует память вне диапазона массива или подобной структуры данных.

Упражнения раздела 3.5.2

Упражнение 3.30. Выявите ошибки индексации в следующем коде

```
constexpr size_t array size = 10;  
int ia[array_size];  
for (size_t ix = 1; ix <= array size; ++ix)  
    ia[ix] = ix;
```

Упражнение 3.31. Напишите программу, где определен массив из десяти целых чисел, каждому элементу которого присвоено значение, соответствующее его позиции в массиве.

Упражнение 3.32. Скопируйте массив, определенный в предыдущем упражнении, в другой массив. Перезапишите эту программу так, чтобы использовались векторы.

Упражнение 3.33. Что будет, если не инициализировать массив `scores` в программе оценок из данного раздела?

3.5.3. Указатели и массивы

Указатели и массивы в языке C++ тесно связаны. В частности, как будет продемонстрировано вскоре, при использовании массивов компилятор обычно преобразует их в указатель.

Обычно указатель на объект получают при помощи оператора обращения к адресу (см. раздел 2.3.2). По правде говоря, оператор обращения к адресу может быть применен к любому объекту, а элементы в массиве — объекты. При индексировании массива результатом является объект в этой области массива. Подобно любым другим объектам, указатель на элемент массива можно получить из адреса этого элемента:

```
string nums[ ] = { "one", "two", "three"}; // массив строк
string *p = &nums[ 0]; // p указывает на первый элемент массива nums
```

Однако у массивов есть одна особенность — места их использования компилятор автоматически заменяет указателем на первый элемент.

```
string *p2 = nums; // эквивалент p2 = &nums[ 0]
```



В большинстве выражений, где используется объект типа массива, в действительности используется указатель на первый элемент в этом массиве.

Существует множество свидетельств того факта, что операции с массивами зачастую являются операциями с указателями. Одно из них — при использовании массива как инициализатора переменной, определенной с использованием спецификатора `auto` (см. раздел 2.5.2), выводится тип указателя, а не массива.

```
int ia[ ] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; // ia - массив из
десяти целых чисел
auto ia2( ia); // ia2 - это int*, указывающий на
первый элемент в ia
ia2 = 42; // ошибка: ia2 - указатель, нельзя
присвоить указателю
// значение типа int
```

Хотя `ia` является массивом из десяти целых чисел, при его использовании в качестве инициализатора компилятор рассматривает это как следующий код:

```
auto ia2(&ia[0]); // теперь ясно, что ia2 имеет тип int*
```

Следует заметить, что это преобразование не происходит, если используется спецификатор `decltype` (см. раздел 2.5.3). Выражение `decltype(ia)` возвращает массив из десяти целых чисел:

```
// ia3 - массив из десяти целых чисел
decltype(ia) ia3 = {0,1,2,3,4,5,6,7,8,9};
ia3 = p;           // ошибка: невозможно присвоить int*
массиву
ia3[4] = i;       // ok: присвоить значение i элементу в
массиве ia3
```

Указатели — это итераторы

Указатели, содержащие адреса элементов в массиве, обладают дополнительными возможностями, кроме описанных в разделе 2.3.2. В частности, указатели на элементы массивов поддерживают те же операции, что и итераторы векторов или строк (см. раздел 3.4). Например, можно использовать оператор инкремента для перемещения с одного элемента массива на следующий:

```
int arr[] = {0,1,2,3,4,5,6,7,8,9};
int *p = arr; // p указывает на первый элемент в arr
++p;          // p указывает на arr[1]
```

Подобно тому, как итераторы можно использовать для перебора элементов вектора, указатели можно использовать для перебора элементов массива. Конечно, для этого нужно получить указатели на первый элемент и элемент, следующий после последнего. Как упоминалось только что, указатель на первый элемент можно получить при помощи самого массива или при обращении к адресу первого элемента. Получить указатель на следующий элемент после последнего можно при помощи другого специального свойства массива. Последний элемент массива `arr` находится в позиции 9, а адрес несуществующего элемента массива, следующего после него, можно получить так:

```
int *e = &arr[10]; // указатель на элемент после
// последнего в массиве arr
```

Единственное, что можно сделать с этим элементом, так это получить

его адрес, чтобы инициализировать указатель `e`. Как и итератор на элемент после конца (см. раздел 3.4.1), указатель на элемент после конца не указывает ни на какой элемент. Поэтому нельзя ни обратиться к его значению, ни прирастить.

Используя эти указатели, можно написать цикл, выводящий элементы массива `arr`.

```
for (int *b = arr; b != e; ++b)
    cout << *b << endl; // вывод элементов arr
```

Библиотечные функции `begin()` и `end()`



Указатель на элемент после конца можно вычислить, но этот подход подвержен ошибкам. Чтобы облегчить и обезопасить использование указателей, новая библиотека предоставляет две функции: `begin()` и `end()`. Эти функции действуют подобно одноименным функциям-членам контейнеров (см. раздел 3.4.1). Однако массивы — не классы, и данные функции не могут быть функциями-членами. Поэтому для работы они получают массив в качестве аргумента.

```
int ia[ ] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; // ia - массив из
десети целых чисел
int *beg = begin(ia); // указатель на первый
элемент массива ia
int *last = end(ia); // указатель на следующий
элемент ia за последним
```

Функция `begin()` возвращает указатель на первый, а функция `end()` на следующий после последнего элемент данного массива. Эти функции определены в заголовке `iterator`.

Используя функции `begin()` и `end()`, довольно просто написать цикл обработки элементов массива. Предположим, например, что массив `arr` содержит значения типа `int`. Первое отрицательное значение в массиве `arr` можно найти следующим образом:

```
// pbegin указывает на первый, а pend на следующий
после последнего
// элемент массива arr
int *pbegin = begin(arr), *pend = end(arr);
// найти первый отрицательный элемент,
остановиться, если просмотрены
```

```
// все элементы
while ( pbeg != pend && *pbeg >= 0)
    ++pbeg;
```

Код начинается с определения двух указателей типа `int` по имени `pbeg` и `pend`. Указатель `pbeg` устанавливается на первый элемент массива `arr`, а `pend` — на следующий элемент после последнего. Условие цикла `while` использует указатель `pend`, чтобы узнать, безопасно ли обращаться к значению указателя `pbeg`. Если указатель `pbeg` действительно указывает на элемент, выполняется проверка результата обращения к его значению на наличие отрицательного значения. Если это так, то условие ложно и цикл завершается. В противном случае указатель переводится на следующий элемент.



Указатель на элемент "после последнего" у встроенного массива ведет себя так же, как итератор, возвращенный функцией `end()` вектора. В частности, нельзя ни обратиться к значению такого указателя, ни осуществить его приращение.

Арифметические действия с указателями

Указатели на элементы массива позволяют использовать все операции с итераторами, перечисленные в табл. 3.6 и 3.7. Эти операции, обращения к значению, инкремента, сравнения, добавления целочисленного значения, вычитания двух указателей, имеют для указателей на элементы встроенного массива то же значение, что и для итераторов.

Результатом добавления (или вычитания) целочисленного значения к указателю (или из него) является новый указатель, указывающий на элемент, расположенный на заданное количество позиций вперед (или назад) от исходного указателя.

```
constexpr size_t sz = 5;
int arr[ sz ] = { 1, 2, 3, 4, 5 };
int *ip = arr;           // эквивалент int *ip = &arr[ 0 ]
int *ip2 = ip + 4;      // ip2 указывает на arr[ 4 ],
последний элемент в arr
```

Результатом добавления 4 к указателю `ip` будет указатель на элемент, расположенный в массиве на четыре позиции далее от того, на который в

настоящее время указывает `i`.

Результатом добавления целочисленного значения к указателю должен быть указатель на элемент (или следующую позицию после конца) в том же массиве:

```
// ok: arr преобразуется в указатель на его первый
// элемент;
// p указывает на позицию после конца arr
int *p = arr + sz; // использовать осмотрительно -
// не обращаться
// к значению!
int *p2 = arr + 10; // ошибка: arr имеет только 5
// элементов;
// значение p2 неопределено
```

При сложении `arr` и `sz` компилятор преобразует `arr` в указатель на первый элемент массива `arr`. При добавлении `sz` к этому указателю получается указатель на позицию `sz` (т.е. на позицию 5) этого массива. Таким образом, он указывает на следующую позицию после конца массива `arr`. Вычисление указателя на более чем одну позицию после последнего элемента является ошибкой, хотя компилятор таких ошибок не обнаруживает.

Подобно итераторам, вычитание двух указателей дает дистанцию между ними. Указатели должны указывать на элементы в том же массиве:

```
auto n = end(arr) - begin(arr); // n - 5,
количество элементов
```

// массива `arr`

Результат вычитания двух указателей имеет библиотечный тип `ptrdiff_t`. Как и тип `size_t`, тип `ptrdiff_t` является машинозависимым типом, определенным в заголовке `cstdint`. Поскольку вычитание способно возвратить отрицательное значение, тип `ptrdiff_t` — знаковый целочисленный.

Для сравнения указателей на элементы (или позицию за концом) массива можно использовать операторы сравнения. Например, элементы массива `arr` можно перебрать следующим образом:

```
int *b = arr, *e = arr + sz;
while (b < e) {
    // используется *b
    ++b;
}
```

Нельзя использовать операторы сравнения для указателей на два несвязанных объекта.

```
int i = 0, sz = 42;
int *p = &i, *e = &sz;
// неопределенно: p и e не связаны; сравнение
бессмысленно!
while (p < e)
```

Хотя на настоящий момент смысл может быть и неясен, но следует заметить, что арифметические действия с указателями допустимы также для нулевых указателей (см. раздел 2.3.2) и для указателей на объекты, не являющиеся массивом. В последнем случае указатели должны указывать на тот же объект или следующий после него. Если *p* — нулевой указатель, то к нему можно добавить (или вычесть) целочисленное константное выражение (см. раздел 2.4.4) со значением 0. Можно также вычесть два нулевых указателя из друг друга, и результатом будет 0.

Взаимодействие обращения к значению с арифметическими действиями с указателями

Результатом добавления целочисленного значения к указателю является указатель. Если полученный указатель указывает на элемент, то к его значению можно обратиться:

```
int ia[] = {0, 2, 4, 6, 8}; // массив из 5 элементов
типа int
int last = *(ia + 4); // ok: инициализирует last
значением
// ia[4], т. е. 8
```

Выражение `*(ia + 4)` вычисляет адрес четвертого элемента после *ia* и обращается к значению полученного указателя. Это выражение эквивалентно выражению `ia[4]`.

Помните, в разделе 3.4.1 обращалось внимание на необходимость круглых скобок в выражениях, содержащих оператор обращения к значению и точечный оператор. Аналогично необходимы круглые скобки вокруг части сложения указателей:

```
last = *ia + 4; // ok: last = 4, эквивалент ia[0] +
4
```

Этот код обращается к значению *ia* и добавляет 4 к полученному значению. Причины подобного поведения рассматриваются в разделе 4.1.2.



Индексирование и указатели

Как уже упоминалось, в большинстве мест, где используется имя массива, в действительности используется указатель на первый элемент этого массива. Одним из мест, где компилятор осуществляет это преобразование, является индексирование массива.

```
int ia[ ] = { 0, 2, 4, 6, 8}; // массив из 5 элементов
типа int
```

Рассмотрим выражение `ia[0]`, использующее имя массива. При индексировании массива в действительности индексируется указатель на элемент в этом массиве.

```
int i = ia[ 2]; // ia преобразуется в указатель на
первый элемент ia
```

```
// ia[ 2 ] выбирает элемент, на
который указывает (ia + 2)
```

```
int *p = ia; // p указывает на первый элемент в
массиве ia
```

```
i = *( p + 2); // эквивалент i = ia[ 2 ]
```

Оператор индексирования можно использовать для любого указателя, пока он указывает на элемент (или позицию после конца) в массиве.

```
int *p = &ia[ 2]; // p указывает на элемент с
индексом 2
```

```
int j = p[ 1]; // p[ 1 ] - эквивалент *( p + 1 ),
// p[ 1 ] тот же элемент, что и
ia[ 3 ]
```

```
int k = p[ -2 ]; // p[ -2 ] тот же элемент, что и
ia[ 0 ]
```

Последний пример указывает на важное отличие между массивами и такими библиотечными типами, как `vector` и `string`, у которых есть операторы индексирования. Библиотечные типы требуют, чтобы используемый индекс был беззнаковым значением. Встроенный оператор индексирования этого не требует. Индекс, используемый со встроенным оператором индексирования, может быть отрицательным значением. Конечно, полученный адрес должен указывать на элемент (или позицию после конца) массива, на который указывает первоначальный указатель.



В отличие от индексов для векторов и строк, индекс встроенного массива не является беззнаковым.

Упражнения раздела 3.5.3

Упражнение 3.34. С учетом, что указатели `p1` и `p2` указывают на элементы в том же массиве, что делает следующий код? Какие значения `p1` или `p2` делают этот код недопустимым?

```
p1 += p2 - p1;
```

Упражнение 3.35. Напишите программу, которая использует указатели для обнуления элементов массива.

Упражнение 3.36. Напишите программу, сравнивающую два массива на равенство. Напишите подобную программу для сравнения двух векторов.

3.5.4. Символьные строки в стиле C



Хотя язык C++ поддерживает строки в стиле C, использовать их в программах C++ не следует. Строки в стиле C — на удивление богатый источник разнообразных ошибок и наиболее распространенная причина проблем защиты.

Символьный строковый литерал — это экземпляр более общей конструкции, которую язык C++ унаследовал от языка C: *символьной строки в стиле C* (C-style character string). Стока в стиле C не является типом данных, скорее это соглашение о представлении и использовании символьных строк. Следующие этому соглашению строки хранятся в символьных массивах и являются строкой с *завершающим нулевым символом* (null-terminated string). Под завершающим нулевым символом подразумевается, что последний видимый символ в строке сопровождается нулевым символом (' \0 '). Для манипулирования этими строками обычно используются указатели.

Строковые функции библиотеки С

Стандартная библиотека языка С предоставляет набор функций, перечисленных в табл. 3.8, для работы со строками в стиле С. Эти функции определены в заголовке `cstring`, являющемся версией С++ заголовка языка С `string.h`.



Функции из табл. 3.8 не проверяют свои строковые параметры

Указатель (указатели), передаваемый этим функциям, должен указывать на массив (массивы) с нулевым символом в конце.

```
char ca[ ] = { 'C', '+', '+' }; // без нулевого
                                // символа в конце
cout << strlen( ca ) << endl; // катастрофа: ca не
                                // завершается нулевым
                                // символом
```

В данном случае `ca` — это массив элементов типа `char`, но он не завершается нулевым символом. Результат непредсказуем. Вероятней всего, функция `strlen()` продолжит просматривать память уже за пределами массива `ca`, пока не встретит нулевой символ.

Таблица 3.8. Функции для символьных строк в стиле С

<code>strlen(p)</code>	Возвращает длину строки <code>p</code> без учета нулевого символа
<code>strcmp(p1, p2)</code>	Проверяет равенство строк <code>p1</code> и <code>p2</code> . Возвращает 0, если <code>p1 == p2</code> , положительное значение, если <code>p1 > p2</code> , и отрицательное значение, если <code>p1 < p2</code>
<code>strcat(p1, p2)</code>	Добавляет строку <code>p2</code> к <code>p1</code> . Результат возвращает в строку <code>p1</code>
<code>strcpy(p1, p2)</code>	Копирует строку <code>p2</code> в строку <code>p1</code> . Результат возвращает в строку <code>p1</code>

Сравнение строк

Сравнение двух строк в стиле С осуществляется совсем не так, как сравнение строк библиотечного типа `string`. При сравнении библиотечных строк используются обычные операторы равенства или сравнения:

```
string s1 = "A string example";
```

```
string s2 = "A different string";
if (s1 < s2) // должно: s2 меньше s1
```

Использование этих же операторов для подобным образом определенных строк в стиле С приведет к сравнению значений указателей, а не самих строк.

```
const char ca1[] = "A string example";
const char ca2[] = "A different string";
if (ca1 < ca2) // непредсказуемо: сравниваются два
адреса
```

Помните, что при использовании массива в действительности используются указатели на их первый элемент (см. раздел 3.5.3). Следовательно, это условие фактически сравнивает два значения `const char*`. Эти указатели содержат адреса разных объектов, поэтому результат такого сравнения непредсказуем.

Чтобы сравнить строки, а не значения указателей, можем использовать функцию `strcmp()`. Она возвращает значение 0, если строки равны, положительное или отрицательное значение, в зависимости от того, больше ли первая строка второй или меньше.

```
if (strcmp(ca1, ca2) < 0) // то же, что и сравнение
строк s1 < s2
```

За размер строки отвечает вызывающая сторона

Конкатенация и копирование строк в стиле С также весьма отличается от таких же операций с библиотечным типом `string`. Например, если необходима конкатенация строк `s1` и `s2`, определенных выше, то это можно сделать так:

```
//      инициализировать      largeStr      результатом
конкатенации строки s1,
// пробела и строки s2
string largeStr = s1 + " " + s2;
```

Подобное с двумя массивами, `ca1` и `ca2`, было бы ошибкой. Выражение `ca1 + ca2` попытается сложить два указателя, что некорректно и бессмысленно.

Вместо этого можно использовать функции `strcat()` и `strcpuy()`. Но чтобы использовать эти функции, им необходимо передать массив для хранения результирующей строки. Передаваемый массив должен быть достаточно большим, чтобы содержать созданную строку, включая нулевой символ в конце. Хотя представленный здесь код следует

традиционной схеме, потенциально он может стать причиной серьезной ошибки.

```
// катастрофа, если размер largeStr вычислен
ошибочно
strcpy(largeStr, ca1); // копирует ca1 в largeStr
strcat(largeStr, " "); // добавляет пробел в конец
largeStr
strcat(largeStr, ca2); // конкатенирует ca2 с
largeStr
```

Проблема в том, что можно легко ошибиться в расчете необходимого размера largeStr. Кроме того, при каждом изменении значения, которые следует сохранить в largeStr, необходимо перепроверить правильность вычисления его размера. К сожалению, код, подобный этому, широко распространен в программах. Такие программы подвержены ошибкам и часто приводят к серьезным проблемам защиты.



Для большинства приложений не только безопасней, но и эффективней использовать библиотечный тип `string`, а не строки в стиле C.

Упражнения раздела 3.5.4

Упражнение 3.37. Что делает следующая программа?

```
const char ca[ ] = { 'h', 'e', 'l', 'l', 'o' };
const char *cp = ca;
while (*cp) {
    cout << *cp << endl;
    ++cp;
}
```

Упражнение 3.38. В этом разделе упоминалось, что не только некорректно, но и бессмысленно пытаться сложить два указателя. Почему сложение двух указателей бессмысленно?

Упражнение 3.39. Напишите программу, сравнивающую две строки. Затем напишите программу, сравнивающую значения двух символьных строк в стиле C.

Упражнение 3.40. Напишите программу, определяющую два символьных массива, инициализированных строковыми литералами. Теперь определите третий символьный массив для содержания результата

конкатенации этих двух массивов. Используйте функции `strcpy()` и `strcat()` для копирования этих двух массивов в третий.

3.5.5. Взаимодействие с устаревшим кодом

Множество программ C++ было написано до появления стандартной библиотеки, поэтому они не используют библиотечные типы `string` и `vector`. Кроме того, многие программы C++ взаимодействуют с программами, написанными на языке С или других языках, которые не могут использовать библиотеку C++. Следовательно, программам, написанным на современном языке C++, вероятно, придется взаимодействовать с кодом, который использует символьные строки в стиле С и/или массивы. Библиотека C++ предоставляет средства, облегчающие такое взаимодействие.



Совместное использование библиотечных строки строк в стиле С

В разделе 3.2.1 была продемонстрирована возможность инициализации строки класса `string` строковым литералом:

```
string s("Hello World"); // s содержит Hello World
```

В общем, символьный массив с нулевым символом в конце можно использовать везде, где используется строковый литерал.

- Символьный массив с нулевым символом в конце можно использовать для инициализации строки класса `string` или присвоения ей.
- Символьный массив с нулевым символом в конце можно использовать как один из операндов (но не оба) в операторе суммы класса `string` или как правый operand в составном операторе присвоения (`+=`) класса `string`.

Однако нет никакого простого способа использовать библиотечную строку там, где требуется строка в стиле С. Например, невозможно инициализировать символьный указатель объектом класса `string`. Тем не менее класс `string` обладает функцией-членом `c_str()`, зачастую позволяющей выполнить желаемое.

```
char *str = s; // ошибка: нельзя инициализировать  
char* из string
```

```
const char *str = s.c_str(); // ok
```

Имя функции `c_str()` означает, что она возвращает символьную

строку в стиле С. Таким образом, она возвращает указатель на начало символьного массива с нулевым символом в конце, содержащим те же символы, что и строка. Тип указателя `const char*` не позволяет изменять содержимое массива.

Допустимость массива, возвращенного функцией `c_str()`, не гарантируется. Любое последующее использование указателя `s`, способное изменить его значение, может сделать этот массив недопустимым.



Если программа нуждается в продолжительном доступе к содержимому массива, возвращенного функцией `c_str()`, то следует создать его копию.

Использование массива для инициализации вектора

В разделе 3.5.1 упоминалось о том, что нельзя инициализировать встроенный массив другим массивом. Инициализировать массив из вектора также нельзя. Однако можно использовать массив для инициализации вектора. Для этого необходимо определить адрес первого подлежащего копированию элемента и элемента, следующего за последним.

```
int int_arr[] = { 0, 1, 2, 3, 4, 5 };
// вектор ivec содержит 6 элементов, каждый из которых является
// копией соответствующего элемента массива int_arr
vector<int> ivec( begin( int_arr ), end( int_arr ) );
```

Два указателя, используемые при создании вектора `ivec`, отмечают диапазон значений, используемых для инициализации его элементов. Второй указатель указывает на следующий элемент после последнего копируемого. В данном случае для передачи указателей на первый и следующий после последнего элементы массива `int_arr` использовались библиотечные функции `begin()` и `end()` (см. раздел 3.5.3). В результате вектор `ivec` содержит шесть элементов, значения которых совпадают со значениями соответствующих элементов массива `int_arr`.

Определяемый диапазон может быть также подмножеством массива:

```
// скопировать 3 элемента: int_arr[1], int_arr[2],
int_arr[3]
vector<int> subVec( int_arr + 1, int_arr + 4 );
```

Этот код создает вектор `subVec` с тремя элементами, значения которых являются копиями значений элементов от `intarr[1]` до `intarr[3]`.

Совет. Используйте вместо массивов библиотечные типы

Указатели и массивы на удивление сильно подвержены ошибкам. Частично проблема в концепции: указатели используются для низкоуровневых манипуляций, в них очень просто сделать тривиальные ошибки. Другие проблемы возникают из-за используемого синтаксиса, особенно синтаксиса объявлений.

Упражнения раздела 3.5.5

Упражнение 3.41. Напишите программу, инициализирующую вектор значениями из массива целых чисел.

Упражнение 3.42. Напишите программу, копирующую вектор целых чисел в массив целых чисел.



3.6. Многомерные массивы

Строго говоря, никаких *многомерных массивов* (multidimensioned array) в языке C++ нет. То, что обычно упоминают как многомерный массив, фактически является массивом массивов. Не забывайте об этом факте, когда будете использовать то, что называют многомерным массивом.

При определении массива, элементы которого являются массивами, указываются две размерности: размерность самого массива и размерность его элементов.

```
int ia[ 3 ][ 4 ]; // массив из 3 элементов; каждый из
которых является
                    // массивом из 4 целых чисел
// массив из 10 элементов, каждый из которых
является массивом из 20
// элементов, каждый из которых является массивом
из 30 целых чисел
int arr[ 10 ][ 20 ][ 30 ] = { 0 }; // инициализировать все
элементы значением 0
```

Как уже упоминалось в разделе 3.5.1, может быть легче понять эти определения, читая их изнутри наружу. Сначала можно заметить определяемое имя, `ia`, далее видно, что это массив размером 3. Продолжая вправо, видим, что у элементов массива `ia` также есть размерность. Таким образом, элементы массива `ia` сами являются массивами размером 4. Глядя влево, видно, что типом этих элементов является `int`. Так, `ia` является массивом из трех элементов, каждый из которых является массивом из четырех целых чисел.

Прочитаем определение массива `arr` таким же образом. Сначала увидим, что `arr` — это массив размером 10 элементов. Элементы этого массива сами являются массивами размером 20 элементов. У каждого из этих массивов по 30 элементов типа `int`. Нет предела количеству используемых индексирований. Поэтому вполне может быть массив, элементы которого являются массивами массив, массив, массив и т.д.

В двумерном массиве первую размерность зачастую называют *рядом* (row), а вторую — *столбцом* (column).

Инициализация элементов многомерного массива

Подобно любым массивам, элементы многомерного массива можно инициализировать, предоставив в фигурных скобках список инициализаторов. Многомерные массивы могут быть инициализированы списками значений в фигурных скобках для каждого ряда.

```
int ia[ 3 ][ 4 ] = { // три элемента; каждый - массив
размером 4
    { 0, 1, 2, 3 }, // инициализаторы ряда 0
    { 4, 5, 6, 7 }, // инициализаторы ряда 1
    { 8, 9, 10, 11 } // инициализаторы ряда 2
};
```

Вложенные фигурные скобки необязательны. Следующая инициализация эквивалентна, хотя и значительно менее очевидна:

```
// эквивалентная инициализация без необязательных
вложенных фигурных
// скобок для каждого ряда
int ia[ 3 ][ 4 ] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
```

Как и в случае одномерных массивов, элементы списка инициализации могут быть пропущены. Следующим образом можно инициализировать только первый элемент каждого ряда:

```
// явная инициализация только нулевого элемента в
каждом ряду
int ia[ 3 ][ 4 ] = { { 0 }, { 4 }, { 8 } };
```

Остальные элементы инициализируются значением по умолчанию, как и обычные одномерные массивы (см. раздел 3.5.1). Но если опустить вложенные фигурные скобки, то результаты были бы совсем иными:

```
// явная инициализация нулевого ряда;
// остальные элементы инициализируются
// по умолчанию
int ix[ 3 ][ 4 ] = { 0, 3, 6, 9 };
```

Этот код инициализирует элементы первого ряда. Остальные элементы инициализируются значением 0.

Индексация многомерных массивов

Подобно любому другому массиву, для доступа к элементам многомерного массива можно использовать индексирование. При этом для каждой размерности используется отдельный индекс.

Если выражение предоставляет столько же индексов, сколько у массива размерностей, получается элемент с определенным типом. Если предоставить меньше индексов, чем есть размерностей, то результатом

будет элемент внутреннего массива по указанному индексу:

```
// присваивает первый элемент массива arr
последнему элементу
// в последнем ряду массива ia
ia[ 2 ][ 3 ] = arr[ 0 ][ 0 ][ 0 ];
int ( &row )[ 4 ] = ia[ 1 ]; // связывает ряд второго
массива с четырьмя
// элементами массива ia
```

В первом примере предоставляются индексы для всех размерностей обоих массивов. Левая часть, `ia[2]`, возвращает последний ряд массива `ia`. Она возвращает не отдельный элемент массива, а сам массив. Индексируем массив, выбирая элемент `[3]`, являющийся последним элементом данного массива.

Точно так же, правый operand имеет три размерности. Сначала выбирается массив по индексу 0 из наиболее удаленного массива. Результат этой операции — массив (многомерный) размером 20. Используя массив размером 30, извлекаем из этого массива с 20 элементами первый элемент. Затем выбирается первый элемент из полученного массива.

Во втором примере `row` определяется как ссылка на массив из четырех целых чисел. Эта ссылка связывается со вторым рядом массива `ia`.

```
constexpr size_t rowCnt = 3, colCnt = 4;
int ia[ rowCnt ][ colCnt ]; // 12 неинициализированных
элементов
// для каждого ряда
for ( size_t i = 0; i != rowCnt; ++i ) {
    // для каждого столбца в ряду
    for ( size_t j = 0; j != colCnt; ++j ) {
        // присвоить элементу его индекс как значение
        ia[ i ][ j ] = i * colCnt + j;
    }
}
```

Внешний цикл `for` перебирает каждый элемент массива `ia`. Внутренний цикл `for` перебирает элементы внутренних массивов. В данном случае каждому элементу присваивается значение его индекса в общем массиве.



Использование серийного оператора `for` с многомерными массивами

По новому стандарту предыдущий цикл можно упростить с помощью серийного оператора `for`:

```
size_t cnt = 0;
for (auto &row : ia)           // для каждого элемента во
внешнем массиве
    for (auto &col : row) { // для каждого элемента во
внутреннем массиве
        col = cnt;           // присвоить значение
текущему элементу
        ++cnt;                // инкремент cnt
    }
```

Этот цикл присваивает элементам массива `ia` те же значения, что и предыдущий цикл, но на сей раз управление индексами берет на себя система. Значения элементов необходимо изменить, поэтому объявляем управляющие переменные `row` и `col` как ссылки (см. раздел 3.2.3). Первый оператор `for` перебирает элементы массива `ia`, являющиеся массивами из 4 элементов. Таким образом, типом `row` будет ссылка на массив из четырех целых чисел. Второй цикл `for` перебирает каждый из этих массивов по 4 элемента. Следовательно, `col` имеет тип `int&`. На каждой итерации значение `cnt` присваивается следующему элементу массива `ia`, а затем осуществляется инкремент переменной `cnt`.

В предыдущем примере как управляющие переменные цикла использовались ссылки, поскольку элементы массива необходимо было изменять. Однако есть и более серьезная причина для использования ссылок. Рассмотрим в качестве примера следующий цикл:

```
for (const auto &row : ia) // для каждого элемента
во внешнем массиве
    for (auto col : row)      // для каждого элемента
во внутреннем массиве
        cout << col << endl;
```

Этому циклу запись в элементы не нужна, но все же управляющая переменная внешнего цикла определена как ссылка. Это сделано для того, чтобы избежать преобразования обычного массива в указатель (см. раздел 3.5.3). Если пренебречь ссылкой и написать эти циклы так, то компиляция потерпит неудачу:

```
for (auto row : ia)
    for (auto col : row)
```

Как и прежде, первый цикл `for` перебирает элементы массива `ia`,

являющиеся массивами по 4 элемента. Поскольку `row` не ссылка, при его инициализации компилятор преобразует каждый элемент массива (как и любой другой объект типа массива) в указатель на первый элемент этого массива. В результате типом `row` в этом цикле будет `int*`. Внутренний цикл `for` некорректен. Несмотря на намерения разработчика, этот цикл пытается перебрать указатель типа `int*`.



Чтобы использовать многомерный массив в серийном операторе `for`, управляющие переменные всех циклов, кроме самого внутреннего, должны быть ссылками.

Указатели и многомерные массивы

Подобно любым другим массивам, имя многомерного массива автоматически преобразуется в указатель на первый его элемент.



Определяя указатель на многомерный массив, помните, что на самом деле он является массивом массивов.

Поскольку многомерный массив в действительности является массивом массивов, тип указателя, в который преобразуется массив, является типом первого внутреннего массива.

```
int ia[ 3 ][ 4 ]; // массив размером 3 элемента; каждый
элемент - массив
                           // из 4 целых чисел
int ( *p )[ 4 ] = ia; // p указывает на массив из
четырех целых чисел
p = &ia[ 2 ];           // теперь p указывает на
последний элемент ia
```

Применяя стратегию из раздела 3.5.1, начнем рассмотрение с части `(*p)`, гласящей, что `p` — указатель. Глядя вправо, замечаем, что объект, на который указывает указатель `p`, имеет размер 4 элемента, а глядя влево, видим, что типом элемента является `int`. Следовательно, `p` — это

указатель на массив из четырех целых чисел.



Круглые скобки в этом объявлении необходимы.

```
int *ip[ 4]; // массив указателей на int
int (*ip)[ 4]; // указатель на массив из четырех
целых чисел
```



Новый стандарт зачастую позволяет избежать необходимости указывать тип указателя на массив за счет использования спецификаторов `auto` и `decltype` (см. раздел 2.5.2).

```
// вывести значение каждого элемента ia; каждый
внутренний массив
// отображается в отдельной строке
// p указывает на массив из четырех целых чисел
for (auto p = ia; p != ia + 3; ++p) {
    // q указывает на первый элемент массива из
    четырех целых чисел;
    // т. е. q указывает на int
    for (auto q = *p; q != *p + 4; ++q)
        cout << *q << ' '; cout << endl;
}
```

Внешний цикл `for` начинается с инициализации указателя `p` адресом первого массива в массиве `ia`. Этот цикл продолжается, пока не будут обработаны все три ряда массива `ia`. Инкремент `++p` перемещает указатель `p` на следующий ряд (т.е. следующий элемент) массива `ia`.

Внутренний цикл `for` выводит значения внутренних массивов. Он начинается с создания указателя `q` на первый элемент в массиве, на который указывает указатель `p`. Результатом `*p` будет массив из четырех целых чисел. Как обычно, при использовании имени массива оно автоматически преобразуется в указатель на его первый элемент. Внутренний цикл `for` выполняется до тех пор, пока не будет обработан каждый элемент во внутреннем массиве. Чтобы получить указатель на элемент сразу за концом внутреннего массива, мы снова обращаемся к

значению указателя *p*, чтобы получить указатель на первый элемент в этом массиве. Затем добавляем к нему 4, чтобы обработать четыре элемента в каждом внутреннем массиве.

Конечно, используя библиотечные функции *begin()* и *end()* (см. раздел 3.5.3), этот цикл можно существенно упростить:

```
// p указывает на первый массив в ia
for (auto p = begin(ia); p != end(ia); ++p) {
    // q указывает на первый элемент во внутреннем
    // массиве
    for (auto q = begin(*p); q != end(*p); ++q)
        cout << *q << ' ';
    // выводит значение,
    // указываемое q
    cout << endl;
}
```

Спецификатор *auto* позволяет библиотеке самостоятельно определить конечный указатель и избавить от необходимости писать тип, значение которого возвращает функция *begin()*. Во внешнем цикле этот тип — указатель на массив из четырех целых чисел. Во внутреннем цикле этот тип — указатель на тип *int*.

Псевдонимы типов упрощают указатели на многомерные массивы

Псевдоним типа (см. раздел 2.5.1) может еще больше облегчить чтение, написание и понимание указателей на многомерные массивы. Рассмотрим пример.

```
using int_array = int[4]; // объявление псевдонима
// типа нового стиля;
// см. раздел 2.5.1
typedef int int_array[4]; // эквивалентное
// объявление typedef;
// см. раздел 2.5.1
// вывести значение каждого элемента ia; каждый
// внутренний массив
// отображается в отдельной строке
for (int_array *p = ia; p != ia + 3; ++p) {
    for (int *q = *p; q != *p + 4; ++q)
        cout << *q << ' ';
    cout << endl;
}
```

Код начинается с определения `int_array` как имени для типа "массив из четырех целых чисел". Это имя типа используется для определения управляющей переменной внешнего цикла `for`.

Упражнения раздела 3.6

Упражнение 3.43. Напишите три разных версии программы для вывода элементов массива `ia`. Одна версия должна использовать для управления перебором серийный оператор `for`, а другие две — обычный цикл `for`, но в одном случае использовать индексирование, а в другом — указатели. Во всех трех программах пишите все типы явно, т.е. не используйте псевдонимы типов и спецификаторы `auto` или `decltype` для упрощения кода.

Упражнение 3.44. Перепишите программы из предыдущего упражнения, используя псевдоним для типа управляющих переменных цикла.

Упражнение 3.45. Перепишите программы снова, на сей раз используя спецификатор `auto`.

Резюме

Одними из важнейших библиотечных типов являются `vector` и `string`. Стока — это последовательность символов переменной длины, а вектор — контейнер объектов единого типа.

Итераторы обеспечивают косвенный доступ к хранящимся в контейнере объектам. Итераторы используются для доступа и перемещения между элементами в строках и векторах.

Массивы и указатели на элементы массива обеспечивают низкоуровневые аналоги библиотечных типов `vector` и `string`. Как правило, предпочтительней использовать библиотечные классы, а не их низкоуровневые альтернативы, массивы и указатели, встроенные в язык.

Термины

Арифметические действия с итераторами (`iterator arithmetic`). Операции с итераторами векторов и строк. Добавление и вычитание целого числа из итератора приводят к изменению позиции итератора на соответствующее количество элементов вперед или назад от исходного. Вычитание двух итераторов позволяет вычислить дистанцию между ними. Арифметические действия допустимы лишь для итераторов, относящихся к

элементам того же контейнера.

Арифметические действия с указателями (pointer arithmetic). Арифметические операции, допустимые для указателей. Указатели на массивы поддерживают те же операции, что и арифметические действия с итераторами.

Индекс (index). Значение, используемое в операторе индексирования для указания элемента, возвращаемого из строки, вектора или массива.

Инициализация значения (value initialization). Инициализация, в ходе которой объекты встроенного типа инициализируются нулем, а объекты класса — при помощи стандартного конструктора класса. Объекты типа класса могут быть инициализированы значением, только если у класса есть стандартный конструктор. Используется при инициализации элементов контейнера, когда указан его размер, но не указан инициализирующий элемент. Элементы инициализируются копией значения, созданного компилятором.

Инициализация копией (copy initialization). Форма инициализации, использующая знак =. Вновь созданный объект является копией предоставленного инициализатора.

Итератор после конца (off-the-end iterator). Итератор, возвращаемый функцией end(). Он указывает не на последний существующий элемент контейнера, а на позицию за его концом, т.е. на несуществующий элемент.

Контейнер (container). Тип, объекты которого способны содержать коллекцию объектов определенного типа. К контейнерным относится тип vector.

Объявление using. Позволяет сделать имя, определенное в пространстве имен, доступным непосредственно в коде. using *пространствоимен*::*имя*; . Теперь имя можно использовать без префикса *пространствоимен*:: .

Оператор! . Оператор логического NOT. Возвращает инверсное значение своего операнда типа bool. Результат true, если operand false, и наоборот.

Оператор&&. Оператор логического AND. Результат true, если оба операнда true. Правый operand обрабатывается, только если левый operand true.

Оператор[]. Оператор индексирования. Оператор obj[i] возвращает элемент в позиции i объекта контейнера obj. Счет индексов начинается с нуля: первый элемент имеет индекс 0, а последний — obj.size() - 1. Индексирование возвращает объект. Если p — указатель, а n — целое

число, то `p[n]` является синонимом для `*(p+n)`.

Оператор | . Оператор логического OR. Результат `true`, если любой операнд `true`. Правый operand обрабатывается, *только если* левый operand `false`.

Оператор++. Для типов итераторов и указателей определен оператор инкремента, который "добавляет единицу", перемещая итератор или указатель на следующий элемент.

Оператор<<. Библиотечный тип `string` определяет оператор вывода, читающий символы в строку.

Оператор->. Оператор стрелка. Объединяет оператор обращения к значению и точечный оператор: `a->b` — синоним для `(*a).b`.

Оператор>>. Библиотечный тип `string` определяет оператор ввода, читающий разграниченные пробелами последовательности символов и сохраняющий их в строковой переменной, указанной правым operandом.

Серийный операторfor (range for). Управляющий оператор, перебирающий значения указанной коллекции и выполняющий некую операцию с каждым из них.

Переполнение буфера (buffer overflow). Грубая ошибка программирования, результат использования индекса, выходящего из диапазона элементов контейнера, такого как `string`, `vector` или массив.

Прямая инициализация (direct initialization). Форма инициализации, не использующая знак `=`.

Расширение компилятора (compiler extension). Дополнительный компонент языка, предлагаемый некоторыми компиляторами. Код, применяющий расширение компилятора, может не подлежать переносу на другие компиляторы.

Создание экземпляра (instantiation). Процесс, в ходе которого компилятор создает специфический экземпляр шаблона класса или функции.

Строка в стиле С (C-style string). Символьный массив с нулевым символом в конце. Строковые литералы являются строками в стиле С. Строки в стиле С могут стать причиной ошибок.

Строка с завершающим нулевым символом (null-terminated string). Стока, последний символ которой сопровождается нулевым символом `(' \0')`.

Типdifference_type. Целочисленный знаковый тип, определенный в классах `vector` и `string`, способный содержать дистанцию между любыми двумя итераторами.

Тип iterator (итератор). Тип, используемый при переборе элементов контейнера и обращении к ним.

Тип ptrdiff_t. Машинозависимый знаковый целочисленный тип, определенный в заголовке `csstddef`. Является достаточно большим, чтобы содержать разницу между двумя указателями в самом большом массиве.

Тип size_t. Машинозависимый беззнаковый целочисленный тип, определенный в заголовке `csstddef`. Является достаточно большим, чтобы содержать размер самого большого возможного массива.

Тип size_type. Имя типа, определенного для классов `vector` и `string`, способного содержать размер любой строки или вектора соответственно. Библиотечные классы, определяющие тип `size_type`, относят его к типу `unsigned`.

Тип string. Библиотечный тип, представляющий последовательность символов.

Тип vector. Библиотечный тип, содержащий коллекцию элементов определенного типа.

Функция begin(). Функция-член классов `vector` и `string`, возвращающая итератор на первый элемент. Кроме того, автономная библиотечная функция, получающая массив и возвращающая указатель на первый элемент в массиве.

Функция empty(). Функция-член классов `vector` и `string`. Возвращает логическое значение (типа `bool`) `true`, если размер нулевой, или значение `false` в противном случае.

Функция end(). Функция-член классов `vector` и `string`, возвращающая итератор на элемент после последнего элемента контейнера. Кроме того, автономная библиотечная функция, получающая массив и возвращающая указатель на элемент после последнего в массиве.

Функция getline(). Определенная в заголовке `string` функция, которой передают поток `istream` и строковую переменную. Функция читает данные из потока до тех пор, пока не встретится символ новой строки, а прочитанное сохраняет в строковой переменной. Функция возвращает поток `istream`. Символ новой строки в прочитанных данных отбрасывается.

Функция push_back(). Функция-член класса `vector`, добавляющая элементы в его конец.

Функция size(). Функция-член классов `vector` и `string` возвращает количество символов или элементов соответственно. Возвращаемое значение имеет тип `size_type` для данного типа.

Шаблон класса (class template). Проект, согласно которому может быть создано множество специализированных классов. Чтобы применить шаблон класса, необходимо указать дополнительную информацию. Например, чтобы определить вектор, указывают тип его элемента: `vector<int>` содержит целые числа.

Глава 4

Выражения

Язык C++ предоставляет богатый набор операторов, а также определяет их назначение и применение к operandам встроенного типа. Он позволяет также определять назначение большинства операторов, operandами которых являются объекты классов. Эта глава посвящена операторам, определенным в самом языке и применяемым к operandам встроенных типов. Будут описаны также некоторые из операторов, определенных библиотекой. Определение операторов для собственных типов рассматривается в главе 14.

Выражение (expression) состоит из одного или нескольких *операторов* (operator) и возвращает *результат* (result) вычисления. Самая простая форма выражения — это одиночной литерал или переменная. Более сложные выражения формируются из оператора и одного или нескольких *operandов* (operand).

4.1. Основы

Существует несколько фундаментальных концепций, определяющих то, как обрабатываются выражения. Начнем с краткого обсуждении концепций, относящихся к большинству (если не ко всем) выражений. В последующих разделах эти темы рассматриваются подробней.



4.1.1. Фундаментальные концепции

Существуют *унарные операторы* (unary operator) и *парные операторы* (binary operator). Унарные операторы, такие как обращение к адресу (`&`) и обращение к значению (`*`), действуют на один операнд. Парные операторы, такие как равенство (`==`) и умножение (`*`), действуют на два операнда. Существует также (всего один) *тройственный оператор* (ternary operator), который использует три операнда, а также *оператор вызова функции* (function call), который получает неограниченное количество операндов.

Некоторые *символы* (symbol), например `*`, используются для обозначения как унарных (обращение к значению), так и парных (умножение) операторов. Представляет ли символ унарный оператор или парный, определяет контекст, в котором он используется. В использовании таких символов нет никакой взаимосвязи, поэтому их можно считать двумя разными символами.

Группировка операторов и operandов

Чтобы лучше понять порядок выполнения выражений с несколькими операторами, следует рассмотреть концепцию *приоритета* (precedence), *порядка* (associativity) и *порядка вычисления* (order of evaluation) операторов. Например, в следующем выражении используются сложение, умножение и деление:

$$5 + 10 * 20 / 2;$$

Операндами оператора `*` могли бы быть числа 10 и 20, либо 10 и `20/2`, либо 15 и 20, либо 15 и `20/2`. Понимание таких выражений является темой следующего раздела.

Преобразование operandов

В ходе вычисления выражения операнды нередко преобразуются из одного типа в другой. Например, парные операторы обычно ожидают operandов одинакового типа. Но операторы применимы и к operandам с разными типами, если они допускают преобразование (см. раздел 2.1.2) в общий тип.

Хотя правила преобразования довольно сложны, по большей части они очевидны. Например, целое число можно преобразовать в число с плавающей запятой, и наоборот, но преобразовать тип указателя в число с плавающей точкой нельзя. Немного неочевидным может быть то, что operandы меньших целочисленных типов (например, `bool`, `char`, `short` и т.д.) обычно *преобразуются* (*promotion*) в больший целочисленный тип, как правило `int`. Более подробная информация о преобразованиях приведена в разделе 4.11.

Перегруженные операторы

Значение операторов для встроенных и составных типов определяет сам язык. Значение большинства операторов типов классов мы можем определить самостоятельно. Поскольку такие определения придают альтернативное значение существующему символу оператора, они называются *перегруженными операторами* (*overloaded operator*). Операторы `>>` и `<<` библиотеки ввода и вывода, а также операторы, использовавшиеся с объектами строк, векторов и итераторов, являются перегруженными операторами.

При использовании перегруженного оператора его смысл, а также тип operandов и результата зависят от того, как определен оператор. Однако количество operandов, их приоритет и порядок не могут быть изменены.



L- и r-значения

Каждое выражение в языке C++ является либо *r-значением* (*r-value*), либо *l-значением* (*l-value*). Эти названия унаследованы от языка С и первоначально имели простую мнемоническую цель: l-значения могли стоять слева от оператора присвоения, а r-значения не могли.

В языке C++ различие не так просто. В языке C++ выражение l-значения возвращает объект или функцию. Однако некоторые l-значения, такие как константные объекты, не могут быть левым operandом присвоения. Кроме того, некоторые выражения возвращают объекты, но

возвращают их как r-, а не l-значения. Короче говоря, при применении объекта в качестве r-значения используется его значение (т.е. его содержимое). При применении объекта в качестве l-значения используется его идентификатор (т.е. его область в памяти).

Операторы различаются по тому, требуют ли они операндов l- или r-значения, а также по тому, возвращают ли они l- или r-значения. Важный момент здесь в том, что (за одним исключением, рассматриваемым в разделе 13.6) l-значение можно использовать там, где требуется r-значение, однако нельзя использовать r-значение там, где требуется l-значение (т.е. область). Когда l-значение применяется вместо r-значения, используется содержимое объекта (его значение). Мы уже использовали несколько операторов, которые задействовали l-значения.

- В качестве своего левого операнда оператор присвоения требует (неконстантного) l-значения и возвращает свой левый operand как l-значение.

- Оператор обращения к адресу (см. раздел 2.3.2) требует в качестве операнда l-значение и возвращает указатель на свой operand как r-значение.

- Встроенные операторы обращения к значению и индексирования (см. раздел 2.3.2 и раздел 3.5.2), а также обращение к значению итератора и операторы индексирования строк и векторов (см. раздел 3.4.1, раздел 3.2.3 и раздел 3.3.3) возвращают l-значения.

- Операторы инкремента и декремента, как встроенные, так и итератора (см. раздел 1.4.1 и раздел 3.4.1), требуют l-значения в качестве operandов. Их префиксные версии (которые использовались до сих пор) также возвращают l-значения.

Рассматривая операторы, следует обратить внимание на то, должен ли operand быть l-значением и возвращает ли он l-значение.

L- и r-значения также различаются при использовании спецификатора `decltype` (см. раздел 2.5.3). При применении спецификатора `decltype` к выражению (отличному от переменной) результатом будет ссылочный тип, если выражение возвращает l-значение. Предположим, например, что указатель `p` имеет тип `int*`. Поскольку обращение к значению возвращает l-значение, выражение `decltype(*p)` имеет тип `int&`. С другой стороны, поскольку оператор обращения к адресу возвращает r-значение, выражение `decltype(&p)` имеет тип `int**`, т.е. указатель на указатель на тип `int`.



4.1.2. Приоритет и порядок

Выражения с двумя или несколькими операторами называются составными (compound expression). Результат составного выражения определяет способ группировки operandов в отдельных операторах. Группировку operandов определяют приоритет и порядок. Таким образом, они определяют, какие части выражения будут operandами для каждого из операторов в выражении. При помощи скобок программисты могут изменять эти правила, обеспечивая необходимую группировку.

Обычно значение выражения зависит от того, как группируются его части. Operandы операторов с более высоким приоритетом группируются прежде operandов операторов с более низким приоритетом. Порядок определяет то, как группируются operandы с тем же приоритетом. Например, операторы умножения и деления имеют одинаковый приоритет относительно друг друга, но их приоритет выше, чем у операторов сложения и вычитания. Поэтому operandы операторов умножения и деления группируются прежде operandов операторов сложения и вычитания. Арифметические операторы имеют левосторонний порядок, т.е. они группируются слева направо.

- Благодаря приоритету результатом выражения $3 + 4 * 5$ будет 23, а не 35.

- Благодаря порядку результатом выражения $20 - 15 - 3$ будет 2, а не 8.

В более сложном примере при вычислении слева направо следующего выражения получается 20:

$6 + 3 * 4 / 2 + 2$

Вполне возможны и другие результаты: 9, 14 и 36. В языке C++ результат составит 14, поскольку это выражение эквивалентно следующему:

```
// здесь круглые скобки соответствуют стандартному  
приоритету и порядку
```

```
((6 + ((3 * 4) / 2)) + 2)
```

Круглые скобки переопределяют приоритет и порядок

Круглые скобки позволяют переопределить обычную группировку. Выражения в круглых скобках обрабатываются как отдельные модули, а во всех остальных случаях применяются обычные правила приоритета.

Например, используя круглые скобки в предыдущем выражении, можно принудительно получить любой из четырех возможных вариантов:

```
// круглые скобки обеспечивают альтернативные  
группировки  
cout << ( 6 + 3) * ( 4 / 2 + 2) << endl; // выводит  
36  
cout << (( 6 + 3) * 4) / 2 + 2 << endl; // выводит  
20  
cout << 6 + 3 * 4 / ( 2 + 2) << endl; // выводит 9
```



Когда важны приоритет и порядок

Мы уже видели примеры, где приоритет влияет на правильность наших программ. Рассмотрим обсуждавшийся в разделе 3.5.3 пример обращения к значению и арифметических действий с указателями.

```
int ia[ ] = { 0, 2, 4, 6, 8}; // массив из 5 элементов  
типа int  
int last = *(ia + 4); // ok: инициализирует last  
значением  
// ia[4], т. е. 8  
last = *ia + 4; // last = 4, эквивалент  
ia[0] + 4
```

Если необходим доступ к элементу в области `ia+4`, то круглые скобки вокруг сложения необходимы. Без круглых скобок сначала группируется часть `*ia`, а к полученному значению добавляется 4.

Наиболее популярный случай, когда порядок имеет значение, — это выражения ввода и вывода. Как будет продемонстрировано в разделе 4.8, операторы ввода и вывода имеют левосторонний порядок. Этот порядок означает, что можно объединить несколько операций ввода и вывода в одном выражении.

```
cin >> v1 >> v2; // читать в v1, а затем в v2
```

В таблице раздела 4.12 перечислены все операторы, организованные по сегментам. У операторов в каждом сегменте одинаковый приоритет, причем сегменты с более высоким приоритетом расположены выше. Например, префиксный оператор инкремента и оператор обращения к значению имеют одинаковый приоритет, который выше, чем таковой у арифметических операторов. Таблица содержит ссылки на разделы, где

описан каждый из операторов. Многие из этих операторов уже применялось, а большинство из остальных рассматривается в данной главе. Подробней некоторые из операторов рассматриваются позже.

Упражнения раздела 4.1.2

Упражнение 4.1. Какое значение возвратит выражение $5 + 10 * 20 / 2$?

Упражнение 4.2. Используя таблицу раздела 4.12, расставьте скобки в следующих выражениях, чтобы обозначить порядок группировки операндов:

- (a) $* \text{vec.begin}()$ (b) $* \text{vec.begin}() + 1$



4.1.3. Порядок вычисления

Приоритет определяет группировку операндов. Но он ничего не говорит о порядке, в котором обрабатываются операнды. В большинстве случаев порядок не определен. В следующем выражении известно, что функции $f1()$ и $f2()$ будут вызваны перед умножением:

```
int i = f1() * f2();
```

В конце концов, умножаются именно их результаты. Тем не менее нет никакого способа узнать, будет ли функция $f1()$ вызвана до функции $f2()$, или наоборот.

Для операторов, которые не определяют порядок вычисления, выражение, пытающееся обратиться к тому же объекту и изменить его, было бы ошибочным. Выражения, которые действительно так поступают, имеют непредсказуемое поведение (см. раздел 2.1.2). Вот простой пример: оператор $<<$ не дает никаких гарантий в том, как и когда обрабатываются его операнды. В результате следующее выражение вывода непредсказуемо:

```
int i = 0;
cout << i << " " << ++i << endl; // непредсказуемо
```

Непредсказуемость этой программы в том, что нет никакой возможности сделать выводы о ее поведении. Компилятор мог бы сначала обработать часть $++i$, а затем часть i , тогда вывод будет 1 1. Но компилятор мог бы сначала обработать часть i , тогда вывод будет 0 1. Либо компилятор мог бы сделать что-то совсем другое. Поскольку у этого выражения неопределенное поведение, программа ошибочна,

независимо от того, какой код создает компилятор.

Четыре оператора действительно гарантируют порядок обработки операндов. В разделе 3.2.3 упоминалось о том, что оператор логического AND (`&&`) гарантирует выполнение сначала левого операнда. Кроме того, он гарантирует, что правый operand обрабатывается только при истинности левого операнда. Другими операторами, гарантирующими порядок обработки operandов, являются оператор логического OR (`||`) (раздел 4.3), условный оператор (`? :`) (раздел 4.7) и оператор запятая (`,`) (раздел 4.10).



Порядок вычисления, приоритет и порядок операторов

Порядок вычисления operandов не зависит от приоритета и порядка операторов. Рассмотрим следующее выражение:

`f() + g() * h() + j()`

- Приоритет гарантирует умножение результатов вызова функций `g()` и `h()`.
- Порядок гарантирует добавление результата вызова функции `f()` к произведению `g()` и `h()`, а также добавление результата сложения к результату вызова функции `j()`.
- Однако нет никаких гарантий относительно порядка вызова этих функций.

Если функции `f()`, `g()`, `h()` и `j()` являются независимыми и не влияют на состояние тех же объектов или выполняют ввод и вывод, то порядок их вызова несуществен. Но если любые из этих функций действительно воздействуют на тот же объект, то выражение ошибочно, а его поведение непредсказуемо.

Упражнения раздела 4.1.3

Упражнение 4.3. Порядок вычисления большинства парных операторов оставляется неопределенным, чтобы предоставить компилятору возможность для оптимизации. Эта стратегия является компромиссом между созданием эффективного кода и потенциальными проблемами в использовании языка программистом. Полагаете этот компромисс приемлемым? Кто-то да, кто-то нет.

Совет. Манипулирование составными выражениями

При написании составных выражений могут пригодиться два эмпирических правила.

1. В сомнительных случаях заключайте выражения в круглые скобки, чтобы явно сгруппировать операнды в соответствии с логикой программы.

2. При изменении значения операнда не используйте этот операнд в другом месте того же оператора.

Важнейшим исключением из второго правила является случай, когда часть выражения, изменяющая operand, сама является operandом другой части выражения. Например, в выражении `*++iter` инкремент изменяет значение итератора `iter`, а измененное значение используется как operand оператора `*`. В этом и подобных выражениях порядок обработки operandов не является проблемным. Но в больших выражениях те части, которые изменяют operand, должны обрабатываться в первую очередь. Такой подход не создает никаких проблем и применяется достаточно часто.

4.2. Арифметические операторы

Таблица 4.1. Арифметические операторы (левосторонний порядок)

Оператор	Действие	Применение
<code>+</code>	Унарный плюс	<code>+ выражение</code>
<code>-</code>	Унарный минус	<code>- выражение</code>
<code>*</code>	Умножение	<code>выражение * выражение</code>
<code>/</code>	Деление	<code>выражение / выражение</code>
<code>%</code>	Остаток	<code>выражение % выражение</code>
<code>+</code>	Сложение	<code>выражение + выражение</code>
<code>-</code>	Вычитание	<code>выражение - выражение</code>

В табл. 4.1 (и таблицах операторов последующих разделов) операторы сгруппированы по приоритету. Унарные арифметические операторы имеют более высокий приоритет, чем операторы умножения и деления, которые в свою очередь имеют более высокий приоритет, чем парные операторы вычитания и сложения. Операторы с более высоким приоритетом группируются перед операторами с более низким приоритетом. Все эти операторы имеют левосторонний порядок, т.е. при равенстве приоритетов они группируются слева направо.

Если не указано иное, то арифметические операторы могут быть применены к любому арифметическому типу (см. раздел 2.1.1) или любому типу, который может быть преобразован в арифметический тип. Операнды и результаты этих операторов являются г-значениями. Как упоминается в разделе 4.11, в ходе вычисления операторов их operandы малых целочисленных типов преобразуются в больший целочисленный тип и все operandы могут быть преобразованы в общий тип.

Унарные операторы плюс и минус могут быть также применены к указателям. Использование парных операторов + и - с указателями рассматривалось в разделе 3.5.3. Будучи примененным к указателю или арифметическому значению, унарный плюс возвращает (возможно, преобразованную) копию значения своего операнда.

Унарный оператор минус возвращает отрицательную копию (возможно, преобразованную) значения своего операнда.

```
int i = 1024;
int k = -i;    // i равно -1024
bool b = true;
bool b2 = -b; // b2 равно true!
```

В разделе 2.1.1 упоминалось, что значения типа `bool` не нужно использовать для вычислений. Результат `-b` — хороший пример того, что имелось в виду.

Для большинства операторов operandы типа `bool` преобразуются в тип `int`. В данном случае значение переменной `b`, `true`, преобразуется в значение 1 типа `int` (см. раздел 2.1.2). Это (преобразованное) значение преобразуется в отрицательное, `-1`. Значение `-1` преобразуется обратно в тип `bool` и используется для инициализации переменной `b2`. Поскольку значение инициализатора отлично от нуля, при преобразовании в тип `bool` его значением станет `true`. Таким образом, значением `b2` будет `true`!

Внимание! Переполнение переменной и другие арифметические особенности

Некоторые арифметические выражения возвращают неопределенный результат. Некоторые из этих неопределенностей имеют математический характер, например деление на нуль. Причиной других являются особенности компьютеров, например, переполнение, происходящее при превышении вычисленным значением размера области памяти, представленной его типом.

Предположим, тип `short` занимает на машине 16 битов. В этом

случае переменная типа `short` способна хранить максимум значение 32767. На такой машине следующий составной оператор присвоения приводит к переполнению.

```
short short_value = 32767; // максимальное
значение при short 16 битов
short_value += 1;          // переполнение
cout << "short_value: " << short_value << endl;
```

Результат присвоения 1 переменной `short_value` непредсказуем. Для хранения знакового значения 32768 требуется 17 битов, но доступно только 16. Многие системы *никак не предупреждают* о переполнении ни во время выполнения, ни во время компиляции. Подобно любой ситуации с неопределенностью, результат оказывается непредсказуем. На системе авторов programma завершилась с таким сообщением:

```
short value: -32768
```

Здесь произошло переполнение переменной: предназначенный для знака разряд содержал значение 0, но был заменен на 1, что привело к появлению отрицательного значения. На другой системе результат мог бы быть иным, либо программа могла бы повести себя по-другому, включая полный отказ.

Примененные к объектам арифметических типов, операторы `+`, `-`, `*` и `/` имеют вполне очевидные значения: сложение, вычитание, умножение и деление. Результатом деления целых чисел является целое число. Получаемая в результате деления дробная часть отбрасывается.

```
int ival1 = 21/6; // ival1 равно 3; результат
усекается
```

```
                                // остаток отбрасывается
int ival2 = 21/7; // ival2 равно 3; остатка нет;
                                // результат - целочисленное
значение
```

Оператор `%` известен как *остаток* (remainder), или *оператор деления по модулю* (modulus). Он позволяет вычислить остаток от деления левого операнда на правый. Его operandы должны иметь целочисленный тип.

```
int ival = 42;
double dval = 3.14;
ival % 12; // ok: возвращает 6
ival % dval; // ошибка: operand с плавающей запятой
```

При делении отличное от нуля частное позитивно, если у операндов одинаковый знак, и отрицательное в противном случае. Прежние версии языка разрешали округление отрицательного частного вверх или вниз; однако новый стандарт требует округления частного до нуля (т.е. усечения).

Оператор деления по модулю определен так, что если m и n целые числа и n отлично от нуля, то $(m/n) * n + m \% n$ равно m . По определению, если $m \% n$ отлично от нуля, то у него тот же знак, что и у m . Прежние версии языка разрешали результату выражения $m \% n$ иметь тот же знак, что и у m , причем на реализациях, у которых отрицательный результат выражения m / n округлялся не до нуля, но такие реализации сейчас запрещены. Кроме того, за исключением сложного случая, где $-m$ приводит к переполнению, $(-m) / n$ и $m / (-n)$ всегда эквивалентны $-(m / n)$, $m \% (-n)$ эквивалентно $m \% n$ и $(-m) \% n$ эквивалентно $-(m \% n)$. А конкретно:

```
21 % 6; /* результат 3 */ 21 / 6; /* результат
3 */
21 % 7; /* результат 0 */ 21 / 7; /* результат
3 */
-21 % -8; /* результат -5 */ -21 / -8; /* результат
2 */
21 % -5; /* результат 1 */ 21 / -5; /* результат
-4 */
```

Упражнения раздела 4.2

Упражнение 4.4. Расставьте скобки в следующем выражении так, чтобы продемонстрировать порядок его обработки. Проверьте свой ответ, откомпилировав и отобразив результат выражения без круглых скобок.

12 / 3 * 4 + 5 * 15 + 24 % 4 / 2

Упражнение 4.5. Определите результат следующих выражений:

- (a) $-30 * 3 + 21 / 5$ (b) $-30 + 3 * 21 / 5$
- (c) $30 / 3 * 21 \% 5$ (d) $-30 / 3 * 21 \% 4$

Упражнение 4.6. Напишите выражение, чтобы определить, является ли значение типа `int` четным или нечетным.

Упражнение 4.7. Что значит переполнение? Представьте три выражения, приводящих к переполнению.

4.3. Логические операторы и операторы отношения

Операторам отношения передают операторы арифметического типа или типа указателя, а логическим операторам — operandы любого типа, допускающего преобразование в тип `bool`. Все они возвращают значение типа `bool`. Арифметические operandы и указатели со значением нуль рассматриваются как значение `false`, а все другие как значение `true`. Operandы для этих операторов являются г-значениями, а результат — г-значение.

Таблица 4.2. Логические операторы и операторы отношения

Порядок	Оператор	Действие	Применение
Правосторонний	!	Логическое NOT	<code>! выражение</code>
Левосторонний	<	Меньше	<code>выражение < выражение</code>
Левосторонний	<=	Меньше или равно	<code>выражение <= выражение</code>
Левосторонний	>	Больше	<code>выражение > выражение</code>
Левосторонний	>=	Больше или равно	<code>выражение >= выражение</code>
Левосторонний	==	Равно	<code>выражение == выражение</code>
Левосторонний	!=	Не равно	<code>выражение != выражение</code>
Левосторонний	&&	Логическое AND	<code>выражение && выражение</code>
Левосторонний		Логическое OR	<code>выражение выражение</code>

Операторы логического AND и OR

Общим результатом оператора логического AND (`&&`) является `true`, если и только если оба его operandы рассматриваются как `true`. Оператор логического OR (`||`) возвращает значение `true`, если любой из его operandов рассматривается как `true`.

Операторы логического AND и OR всегда обрабатывают свой левый operand перед правым. Кроме того, правый operand обрабатывается, *если и только если* левый operand не определил результат. Эта стратегия известна как *вычисление по сокращенной схеме* (short-circuit evaluation).

- Правая сторона оператора `&&` вычисляется, если и только если левая сторона истинна.
- Правая сторона оператора `||` вычисляется, если и только если левая сторона ложна.

Оператор логического AND использовался в некоторых из программ главы 3. Эти программы использовали левый operand для проверки, безопасно ли выполнять правый operand. Например, условие цикла `for` в разд 3.2.3: сначала проверялось, что `index` не достиг конца строки:

```
index != s.size() && ! isspace(s[index])
```

Это гарантировало, что правый operand не будет выполнен, если индекс уже вышел из диапазона.

Рассмотрим пример применения оператора логического OR. Предположим, что в векторе строк имеется некоторый текст, который необходимо вывести, добавляя символ новой строки после каждой пустой строки или после строки, завершающейся точкой. Для отображения каждого элемента используем серийный оператор `for` (раздел 3.2.3):

```
// обратите внимание, s - ссылка на константу;
элементы не копируются и
// не могут быть изменены
for (const auto &s : text) { // для каждого
элемента text
    cout << s; // вывести текущий
элемент
    // пустые строки и строки, завершающиеся точкой,
требуют новой строки
    if (s.empty() || s[s.size() - 1] == '.')
        cout << endl;
    else
        cout << " "; // в противном случае отделить
пробелом
}
```

После вывода текущего элемента выясняется, есть ли необходимость выводить новую строку. Условие оператора `if` сначала проверяет, не пуста ли строка `s`. Если это так, то необходимо вывести новую строку независимо от значения правого операнда. Только если строка не пуста, обрабатывается второе выражение, которое проверяет, не заканчивается ли строка точкой. Это выражение полагается на вычисление по сокращенной схеме оператора `||`, гарантирующего индексирование строки `s`, только если она не пуста.

Следует заметить, что переменная `s` объявлена как ссылка на константу (см. раздел 2.5.2). Элементами вектора `text` являются строки, и они могут быть очень большими, а использование ссылки позволяет избежать их копирования. Поскольку запись в элементы не нужна, объявляем `s` ссылкой на константу.

Оператор логического NOT

Оператор логического NOT (!) возвращает инверсию исходного значения своего операнда. Этот оператор уже использовался в разделе 3.2.2. В следующем примере подразумевается, что `vec` — это вектор целых чисел, для проверки наличия значений в элементах которого используется оператор логического NOT для значения, возвращенного функцией `empty()`.

```
// отобразить первый элемент вектора vec, если он есть
```

```
if (!vec.empty())
    cout << vec[0];
```

Подвыражение `!vec.empty()` возвращает значение `true`, если вызов функции `empty()` возвращает значение `false`.

Операторы отношения

Операторы отношения (`<`, `<=`, `>`, `>=`) имеют свой обычный смысл и возвращают значение типа `bool`. Эти операторы имеют левосторонний порядок.

Поскольку операторы отношения возвращают логическое значение, их сцепление может дать удивительный результат:

```
// Упс! Это условие сравнивает k с результатом сравнения i < j
if (i < j < k) // true, если k больше 1!
```

Условие группирует `i` и `j` в первый оператор `<`. Результат этого выражения (типа `bool`) является левым операндом второго оператора `<`. Таким образом, переменная `k` сравнивается с результатом (`true` или `false`) первого оператора сравнения! Для реализации той проверки, которая и предполагалась, выражение нужно переписать следующим образом:

```
// условие истинно, если i меньше, чем j, и j меньше, чем k
if (i < j && j < k) { /* ... */ }
```

Проверка равенства и логические литералы

Если необходимо проверить истинность арифметического значения или объекта указателя, то самый простой способ подразумевает использование этого значения как условия.

```
if (val) { /* ... */ } // true, если val - любое не нулевое значение
```

```
if (!val) { /* ... */ } // true, если val - нуль
```

В обоих условиях компилятор преобразовывает `val` в тип `bool`. Первое условие истинно, пока значение переменной `val` отлично от нуля; второе истинно, если `val` — нуль.

Казалось бы, условие можно переписать так:

```
if (val == true) { /* ... */ } // true, только если  
val равно 1!
```

У этого подхода две проблемы. Прежде всего, он длинней и менее непосредствен, чем предыдущий код (хотя по общему признанию в начале изучения языка C++ этот код понятней). Но важней всего то, что если тип переменной `val` отличен от `bool`, то это сравнение работает не так, как ожидалось.

Если переменная `val` имеет тип, отличный от `bool`, то перед применением оператора `==` значение `true` преобразуется в тип переменной `val`. Таким образом, получается код, аналогичный следующему:

```
if (val == 1) { /*...*/ }
```

Как уже упоминалось, при преобразовании значения типа `bool` в другой арифметический тип `false` преобразуется в 0, а `true` — в 1 (см. раздел 2.1.2). Если бы нужно было действительно сравнить значение переменной `val` со значением 1, то условие так и следовало бы написать.



Использование логических литералов `true` и `false` в качестве операндов сравнения — обычно плохая идея. Эти литералы следует использовать только для сравнения с объектами типа `bool`.

Упражнения раздела 4.3

Упражнение 4.8. Объясните, когда обрабатываются операнды операторов логического AND, логического OR и оператора равенства.

Упражнение 4.9. Объясните поведение следующего условия оператора `if`:

```
const char *cp = "Hello World";  
if (cp && *cp)
```

Упражнение 4.10. Напишите условие цикла `while`, который читал бы целые числа со стандартного устройства ввода, пока во вводе не встретится

значение 42.

Упражнение 4.11. Напишите выражение, проверяющее четыре значения *a*, *b*, *c* и *d* и являющееся истинным, если значение *a* больше *b*, которое больше *c*, которое больше *d*.

Упражнение 4.12. С учетом того, что *i*, *j* и *k* имеют тип *int*, объясните значение выражения *i != j < k*.

4.4. Операторы присвоения

Левым операндом оператора присвоения должно быть допускающее изменение l-значение. Ниже приведено несколько примеров недопустимых попыток присвоения.

```
int i = 0, j = 0, k = 0; // инициализация, а не присвоение
const int ci = i;           // инициализация, а не присвоение
1024 = k; // ошибка: литерал является r-значением
i + j = k; // ошибка: арифметическое выражение - тоже r-значение
ci = k;    // ошибка: ci - константа (неизменяемое l-значение)
```

Результат присвоения, левый операнд, является l-значением. Тип результата совпадает с типом левого операнда. Если типы левого и правого операндов отличаются, тип правого операнда преобразуется в тип левого.

```
k = 0;          // результат: тип int, значение 0
k = 3.14159; // результат: тип int, значение 3
```



По новому стандарту с правой стороны можно использовать список инициализации (см. раздел 2.2.1):

```
k = { 3.14 };                                // ошибка: сужающее преобразование
vector<int> vi;                            // первоначально пусто
vi = { 0,1,2,3,4,5,6,7,8,9 }; // теперь vi содержит десять элементов
                                    // со значениями от 0 до 9
```

Если левый operand имеет встроенный тип, список инициализации

может содержать максимум одно значение, и это значение не должно требовать *сужающего преобразования* (narrowing conversion) (см. раздел 2.2.1).

Для типов классов происходящее зависит от подробностей класса. В случае вектора шаблон `vector` определяет собственную версию оператора присвоения, позволяющего использовать список инициализации. Этот оператор заменяет элементы вектора с левой стороны элементами списка с правой.

Независимо от типа левого операнда список инициализации может быть пуст. В данном случае компилятор создает инициализированный значением по умолчанию (см. раздел 3.3.1) временный объект и присваивает это значение левому операнду.

Оператор присвоения имеет правосторонний порядок

В отличие от других парных операторов, присвоение имеет правосторонний порядок:

```
int ival, jval;  
ival = jval = 0; // ok: каждой переменной присвоено  
значение 0
```

Поскольку присвоение имеет правосторонний порядок, его крайняя правая часть, `jval = 0`, является правым operandом крайнего левого оператора присвоения. Поскольку присвоение возвращает свой левый operand, результат крайнего правого присвоения (т.е. `jval`) присваивается переменной `ival`.

Каждый объект в множественном операторе присвоения должен иметь тип, совпадающий с типом соседа справа, или допускать преобразование в него (раздел 4.11):

```
int ival, *pval; // ival имеет тип int; pval имеет  
типа указателя на int  
ival = pval = 0; // ошибка: переменной типа int  
нельзя присвоить  
                                // значение указателя  
string s1, s2;  
s1 = s2 = "OK";      // строковый литерал "OK"  
преобразован в строку
```

Первое присвоение некорректно, поскольку объекты `ival` и `pval` имеют разные типы и не существует преобразования типа `int*` (`pval`) в тип `int` (`ival`). Оно некорректно, несмотря на то, что значение нуль

может быть присвоено любому объекту.

Второе присвоение, напротив, вполне допустимо. Строковый литерал преобразуется в значение типа `string`, которое и присваивается переменной `s2` типа `string`. Результат этого присвоения — строка `s2` — имеет тот же тип, что и строка `s1`.

Оператор присвоения имеет низкий приоритет

Присвоения нередко происходят в условиях. Поскольку оператор присвоения имеет относительно низкий приоритет, его обычно заключают в скобки, чтобы он работал правильно. Чтобы продемонстрировать, чем присвоение может быть полезно в условии, рассмотрим следующий цикл. Здесь необходимо вызывать функцию до тех пор, пока она не возвратит желаемое значение, скажем 42.

```
// подробный, а потому более подверженный ошибкам
// способ написания цикла
int i = get_value(); // получить первое значение
while (i != 42) {
    // выполнить действия ...
    i = get_value(); // получить остальные значения
}
```

Код начинается с вызова функции `get_value()`, затем следует цикл, условие которого использует значение, возвращенное этим вызовом. Последним оператором этого цикла является еще один вызов функции `get_value()`, далее цикл повторяется. Этот код можно переписать более непосредственно:

```
int i;
// лучший вариант цикла, теперь вполне понятно, что
делает условие
while ((i = get_value()) != 42) {
    // выполнить действия ...
}
```

Теперь условие вполне однозначно выражает намерение разработчика: необходимо продолжать, пока функция `get_value()` не возвратит значение 42. В ходе вычисления условия результат вызова функции `get_value()` присваивается переменной `i`, значение которой затем сравнивается со значением 42.

Без круглых скобок операндами оператора `!=` было бы значение, возвращенное функцией `get_value()` и 42, а результат проверки (`true`

или `false`) был бы присвоен переменной `i`, чего явно не планировалось!



Поскольку приоритет оператора присвоения ниже, чем у операторов отношения, круглые скобки вокруг присвоений в условиях обычно необходимы.

Не перепутайте операторы равенства и присвоения

Тот факт, что присвоение возможно в условии, зачастую имеет удивительные последствия:

```
if (i = j)
```

Условие оператора `if` присваивает значение переменной `j` переменной `i`, а затем проверяет результат присвоения. Если значение переменной `j` отлично от нуля, то условие истинно. Однако автор этого кода почти наверняка намеревался проверить равенство значений переменных `i` и `j` так:

```
if (i == j)
```

Ошибки такого рода хоть и известны, но трудны для обнаружения. Некоторые, но не все компиляторы достаточно "любезны", чтобы предупредить о таком коде, как в этом примере.

Составные операторы присвоения

Довольно нередки случаи, когда оператор применяется к объекту, а полученный результат повторно присваивается тому же объекту. В качестве примера рассмотрим программу из раздела 1.4.2:

```
int sum = 0;  
// сложить числа от 1 до 10 включительно  
for (int val = 1; val <= 10; ++val)  
    sum += val; // эквивалентно sum = sum + val
```

Подобный вид операций характерен не только для сложения, но и для других арифметических и побитовых операторов, которые рассматриваются в разделе 4.8. Соответствующие составные операторы присвоения (compound assignment) существуют для каждого из этих операторов.

```
+ = - = * = / = %= // арифметические операторы  
<<= >>= & = ^ = | = // побитовые операторы; см. р. 4.8
```

Каждый составной оператор по сути эквивалентен обычному, за исключением того, что, когда используется составное присвоение, левый операнд обрабатывается (оценивается) только однажды.

Но эти формы имеют одно очень важное различие: в составном операторе присвоения левый операнд вычисляется только один раз. По существу, он эквивалентен следующему:

`a = a operator b;`

Если используется обычное присвоение, операнд `a` обрабатывается дважды: один раз в выражении с правой стороны и во второй раз — как операнд слева. В подавляющем большинстве случаев это различие несущественно, возможно, кроме тех, где критически важна производительность.

Упражнения раздела 4.4

Упражнение 4.13. Каковы значения переменных `i` и `d` после каждого присвоения?

`int i; double d;`
(a) `d = i = 3.5;` (b) `i = d = 3.5;`

Упражнение 4.14. Объясните, что происходит в каждом из следующих операторов `if`?

`if (42 = i) // ...`
`if (i = 42) // ...`

Упражнение 4.15. Следующее присвоение недопустимо. Почему? Как исправить ситуацию?

`double dval; int ival; int *pi;`
`dval = ival = pi = 0;`

Упражнение 4.16. Хотя ниже приведены вполне допустимые выражения, их поведение может оказаться не таким, как предполагалось. Почему? Перепишите выражения так, чтобы они стали более понятными.

(a) `if (p = getPtr() != 0)`
(b) `if (i = 1024)`

4.5. Операторы инкремента и декремента

Операторы инкремента (`++`) и декремента (`--`) позволяют в краткой и удобной форме добавить или вычесть единицу из объекта. Эта форма записи обеспечивает не только удобство, она весьма популярна при работе с итераторами, поскольку большинство итераторов не поддерживает арифметических действий.

Эти операторы существуют в двух формах: префиксной и постфиксной. До сих пор использовался только *префиксный оператор инкремента* (prefix increment). Он осуществляет инкремент (или декремент) своего операнда и возвращает измененный объект как результат. *Постфиксный оператор инкремента* (postfix increment) (или декремента) возвращает копию первоначального операнда *неизменной*, а затем изменяет значение операнда.

```
int i = 0, j;  
j = ++i; // j = 1, i = 1: префикс возвращает  
увеличенное значение  
j = i++; // j = 1, i = 2: постфикс возвращает  
исходное значение
```

Операндами этих операторов должны быть l-значения. Префиксные операторы возвращают сам объект как l-значение. Постфиксные операторы возвращают копию исходного значения объекта как r-значение.

***Совет. Используйте постфиксные операторы только по
мере необходимости***

Читатели с опытом языка C могли бы быть удивлены тем, что в написанных до сих пор программах использовался префиксный оператор инкремента. Причина проста: префиксная версия позволяет избежать ненужной работы. Она увеличивает значение и возвращает результат. Постфиксный оператор должен хранить исходное значение, чтобы возвратить неувеличенное значение как результат. Но если в исходном значении нет никакой потребности, то нет необходимости и в дополнительных действиях, осуществляемых постфиксным оператором.

Для переменных типа `int` и указателей компилятор способен оптимизировать код и уменьшить количество дополнительных действий. Для более сложных типов итераторов подобные дополнительные действия могут обойтись довольно дорого. При использовании префиксных версий об эффективности можно не волноваться. Кроме того, а возможно и важнее всего то, что так можно выразить свои намерения более непосредственно.



Объединение операторов обращения к значению и инкремента в одном выражении

Постфиксные версии операторов `++` и `--` используются в случае, когда в одном составном выражении необходимо использовать текущее значение переменной, а затем увеличить его.

В качестве примера используем постфиксный оператор инкремента для написания цикла, выводящего значения вектора до, но не включая, первого отрицательного значения.

```
auto pbegin = v.begin();
// отображать элементы до первого отрицательного
значения
while (pbegin != v.end() && *pbegin >= 0)
    cout << *pbegin++ << endl; // отобразить текущее
значение и
                                // переместить указатель
pbegin
```

Выражение `*pbegin++` обычно малопонятно новичкам в языках C++ и C. Но поскольку эта схема весьма распространена, программисты C++ должны понимать такие выражения.

Приоритет постфиксного оператора инкремента выше, чем оператора обращения к значению, поэтому код `*pbegin++` эквивалентен коду `*(pbegin++)`. Часть `pbegin++` осуществляет инкремент указателя `pbegin` и возвращает как результат копию предыдущего значения указателя `pbegin`. Таким образом, операндом оператора `*` будет неувеличенное значение указателя `pbegin`. Следовательно, оператор выводит элемент, на который первоначально указывал указатель `pbegin`, а затем осуществляет его инкремент.

Этот подход основан на том, что постфиксный оператор инкремента возвращает копию своего исходного, не увеличенного операнда. Если бы он возвратил увеличенное значение, то обращение к элементу вектора по такому увеличенному значению привело бы к плачевным результатам: первым оказался бы незаписанный элемент вектора. Хуже того, если бы у последовательности не было никаких отрицательных значений, то в конце произошла бы попытка обращения к значению несуществующего элемента за концом вектора.

Совет. Краткость может быть достоинством

Такие выражения, как `*iter++`, могут быть не очевидны, однако они весьма популярны. Следующая форма записи проще и менее подвержена ошибкам:

```
cout << *iter++ << endl;  
чем ее более подробный эквивалент:  
cout << *iter << endl;  
++iter;
```

Поэтому примеры подобного кода имеет смысл внимательно изучать, чтобы они стали совершенно понятны. В большинстве программ C++ используются краткие формы выражений, а не их более подробные эквиваленты. Поэтому программистам C++ придется привыкать к ним. Кроме того, научившись работать с краткими формами, можно заметить, что они существенно менее подвержены ошибкам.

Помните, что операнды могут быть обработаны в любом порядке

Большинство операторов не гарантирует последовательности обработки operandов (см. раздел 4.1.3). Отсутствие гарантированного порядка зачастую не имеет значения. Это действительно имеет значение в случае, когда выражение одного операнда изменяет значение, используемое выражением другого. Поскольку операторы инкремента и декремента изменяют свои operandы, очень просто неправильно использовать эти операторы в составных выражениях.

Для иллюстрации проблемы перепишем цикл из раздела 3.4.1, который преобразует в верхний регистр символы первого введенного слова:

```
for (auto it = s.begin(); it != s.end() &&  
!isspace(*it); ++it)  
    it = toupper(*it); // преобразовать в верхний  
регистр
```

Этот пример использует цикл `for`, позволяющий отделить оператор обращения к значению `beg` от оператора его приращения. Замена цикла `for`, казалось бы, эквивалентным циклом `while` дает неопределенные результаты:

```
// поведение следующего цикла неопределено!  
while (beg != s.end() && !isspace(*beg))  
    beg = toupper(*beg++); // ошибка: это присвоение  
неопределено
```

Проблема пересмотренной версии в том, что левый и правый operandы оператора `=` используют значение, на которое указывает `beg`, и правый его изменяет. Поэтому присвоение неопределено. Компилятор мог бы обработать это выражение так:

```
*beg = toupper(*beg); // сначала
```

обрабатывается левая сторона
*(beg + 1) = toupper(*beg); // сначала
обрабатывается правая сторона
Или любым другим способом.

Упражнения раздела 4.5

Упражнение 4.17. Объясните различие между префиксным и постфиксным инкрементом.

Упражнение 4.18. Что будет, если цикл `while` из последнего пункта этого раздела, используемый для отображения элементов вектора, задействует префиксный оператор инкремента?

Упражнение 4.19. С учетом того, что `ptr` указывает на тип `int`, `vec` — вектор `vector<int>`, а `ival` имеет тип `int`, объясните поведение каждого из следующих выражений. Есть ли среди них неправильные? Почему? Как их исправить?

- (a) `ptr != 0 && *ptr++` (b) `ival++ && ival`
- (c) `vec[ival++] <= vec[ival]`

4.6. Операторы доступа к членам

Операторы точка (`.`) (dot operator) (см. раздел 1.5.2) и стрелка (`->`) (arrow operator) (см. раздел 3.4.1) обеспечивают доступ к члену. Оператор точка выбирает член из объекта типа класса; оператор стрелка определен так, что код `ptr->mem` эквивалентен коду `(*ptr).mem`.

```
string s1 = "a string", *p = &s1;
auto n = s1.size(); // вызов функции-члена size()
строки s1
n = (*p).size(); // вызов функции-члена size()
объекта, на который
                                // указывает указатель p
n = p->size(); // эквивалент (*p).size()
```

Поскольку приоритет обращения к значению ниже, чем оператора точки, часть обращения к значению следует заключить в скобки. Если пропустить круглые скобки, этот код поведет себя совсем по-иному:

```
// вызов функции-члена size() объекта, на который
указывает указатель p
// затем обращение к значению результата!
*p.size(); // ошибка: p - указатель, он не имеет
функции-члена size()
```

Этот код пытается вызвать функцию-член `size()` объекта `p`. Однако `p` — это указатель, у которого нет никаких членов; этот код не будет откомпилирован.

Оператор стрелка получает операнд в виде указателя и возвращает l-значение. Оператор точки возвращает l-значение, если объект, член которого выбирается, является l-значением; в противном случае результат — r-значение.

Упражнения раздела 4.6

Упражнение 4.20. С учетом того, что `iter` имеет тип `vector<string>::iterator`, укажите, какие из следующих выражений допустимы, если таковые имеются. Объясните поведение допустимых выражений, и почему ошибочные не допустимы?

- (a) `*iter++;` (b) `(*iter)++;` (c) `*iter.empty()`
- (d) `iter->empty();` (e) `++*iter;` (f) `iter++->empty();`

4.7. Условный оператор

Условный оператор (оператор `? :`) (conditional operator) позволяет внедрять простые конструкции `if...else` непосредственно в выражение. Условный оператор имеет следующий синтаксис:

условие ? *выражение1* : *выражение2*;

где *условие* — это выражение, используемое в качестве условия, а *выражение1* и *выражение2* — это выражения одинаковых типов (или типов, допускающих преобразование в общий тип). Эти выражения выполняются в зависимости от *условия*. Если условие истинно, то выполняется *выражение1*; в противном случае выполняется *выражение2*. В качестве примера использования условного оператора рассмотрим код, определяющий, является ли оценка (`grade`) проходной (`pass`) или нет (`fail`):

```
string finalgrade = (grade < 60) ? "fail" : "pass";
```

Условие проверяет, не меньше ли оценка 60. Если это так, то результат выражения `"fail"`; в противном случае — результат `"pass"`. Подобно операторам логического AND и OR (`&&` и `||`), условный оператор гарантирует, что выполнено будет только одно из выражений, *выражение1* или *выражение2*.

Результат условного оператора — l-значение, если оба выражения l-

значения или если они допускают преобразование в общий тип l-значения. В противном случае результат — r-значение.

Вложенные условные операторы

Один условный оператор можно вложить в другой. Таким образом, условный оператор применяются как *условие* или как один или оба *выражения* другого условного оператора. В качестве примера используем пару вложенных условных операторов для трехступенчатой проверки оценки, чтобы выяснить, является ли она выше проходной, просто проходной или непроходной.

```
finalgrade = ( grade > 90) ? "high pass"  
                      : ( grade < 60) ? "fail" :  
"pass";
```

Первое условие проверяет, не выше ли оценка 90. Если это так, то выполняется выражение после ?, возвращающее литерал "high pass". Если условие ложно, выполняется ветвь :, которая сама является другим условным выражением. Это условное выражение проверяет, не меньше ли оценка 60. Если это так, то обрабатывается ветвь ?, возвращающая литерал "fail". В противном случае ветвь : возвращает литерал "pass".

Условный оператор имеет правосторонний порядок, т.е. его operandы группируются (как обычно) справа налево. Порядок объясняет тот факт, что правое условное выражение, сравнивающее grade со значением 60, образует ветвь : левого условного выражения.



Вложенные условные выражения быстро становятся нечитабельными, поэтому нежелательно создавать больше двух или трех вложений.

Применение условного оператора в выражении вывода

Условный оператор имеет довольно низкий приоритет. Когда условный оператор используется в большом выражении, его, как правило, следует заключать в круглые скобки. Например, условный оператор нередко используют для отображения одного из значений в зависимости от результата проверки условия. Отсутствие круглых скобок вокруг условного оператора в выражении вывода может привести к неожиданным

результатам:

```
cout << ((grade < 60) ? "fail" : "pass"); //  
выводит pass или fail  
cout << (grade < 60) ? "fail" : "pass"; //  
выводит 1 или 0!  
cout << grade < 60 ? "fail" : "pass"; // ошибка:  
сравнивает cout с 60
```

Второе выражение использует сравнение `grade` и `60` как operand оператора `<<`. В зависимости от истинности или ложности выражения `grade < 60` выводится значение `1` или `0`. Оператор `<<` возвращает объект `cout`, который и проверяется в условии условного оператора. Таким образом, второе выражение эквивалентно следующему:

```
cout << (grade < 60); // выводит 1 или 0  
cout ? "fail" : "pass"; // проверяет cout, а затем  
возвращает один из  
// этих двух литералов в  
зависимости от  
// истинности объекта cout
```

Последнее выражение ошибочно, поскольку оно эквивалентно следующему:

```
cout << grade; // приоритет оператора ниже, чем у  
// сдвига, поэтому сначала выводится  
оценка,  
cout < 60 ? "fail" : "pass"; // затем cout  
сравнивается с 60!
```

Упражнения раздела 4.7

Упражнение 4.21. Напишите программу, использующую условный оператор для поиска в векторе `vector<int>` элементов с нечетным значением и их удвоения.

Упражнение 4.22. Дополните программу, присваивающую переменной значение оценки (высокая, проходная, не проходная), еще одной оценки, минимально проходной, от `60` до `75` включительно. Напишите две версии: одна использует только условные операторы; вторая использует один или несколько операторов `if`. Как по вашему, какую версию проще понять и почему?

Упражнение 4.23. Следующее выражение не компилируется из-за приоритета операторов. Используя таблицу из раздела 4.12, объясните причину проблемы. Как ее исправить?

```
string s = "word";
string p1 = s + s[ s.size( ) - 1] == 's' ? "" : "s" ;
```

Упражнение 4.24. Программа, различавшая проходную и непроходную оценку, зависела от того факта, что условный оператор имеет правосторонний порядок. Опишите, как обрабатывался бы этот оператор, имей он левосторонний порядок.

4.8. Побитовые операторы

Побитовые операторы (bitwise operator) получают операнды целочисленного типа, которые они используют как коллекции битов. Эти операторы позволяют проверять и устанавливать отдельные биты. Как будет описано в разделе 17.2, эти операторы можно также использовать для библиотечного типа `bitset`, представляющего коллекцию битов изменяемого размера.

Как обычно, если operand — "малое целое число", его значение сначала преобразуется (раздел 4.11) в больший целочисленный тип. Операнды могут быть знаковыми или беззнаковыми.

Таблица 4.3. Побитовые операторы (левосторонний порядок)

Оператор	Действие	Применение
<code>~</code>	Побитовое NOT	<code>~ выражение</code>
<code><<</code>	Сдвиг влево	<code>выражение1 << выражение2</code>
<code>>></code>	Сдвиг вправо	<code>выражение1 >> выражение2</code>
<code>&</code>	Побитовое AND	<code>выражение1 & выражение2</code>
<code>^</code>	Побитовое XOR	<code>выражение1 ^ выражение2</code>
<code> </code>	Побитовое OR	<code>выражение1 выражение2</code>

Если operand знаковый и имеет отрицательное значение, то способ обработки "знакового разряда" большинства битовых операций зависит от конкретной машины. Кроме того, результат сдвига влево, изменяющего знаковый разряд, непредсказуем.



Поскольку нет никаких гарантий однозначного выполнения побитовых операторов со знаковыми переменными на разных машинах, настоятельно рекомендуется использовать в них только беззнаковые целочисленные значения.

Побитовые операторы сдвига

Мы уже использовали перегруженные версии операторов `>>` и `<<`, которые библиотека IO определяет для ввода и вывода. Однако первоначальное значение этих операторов — побитовый сдвиг operandов.

Они возвращают значение, являющееся копией (возможно преобразованной) левого операнда, биты которого сдвинуты. Правый operand не должен быть отрицательным, и его значение должно быть меньше количества битов результата. В противном случае операция имеет неопределенный результат. Биты сдвигаются влево (`<<`) или право (`>>`), при этом вышедшие за пределы биты отбрасываются.

Оператор сдвига влево (`<<`) (left-shift operator) добавляет нулевые биты справа. Поведение *оператора сдвига вправо* (`>>`) (right-shift operator) зависит от типа левого операнда: если он беззнаковый, то оператор добавляет слева нулевые биты; если он знаковый, то результат зависит от конкретной реализации: слева вставляются либо копии знакового разряда, либо нули.

В этих примерах подразумевается, что младший бит расположен справа, тип `char` содержит 8 битов, а тип `int` — 32 бита

```
// 0233 — восьмеричный литерал (см. раздел 2.1.3)
unsigned char bits = 0233; 1 0 0 1 1 0 1 1
bits << 8 // bits преобразуется в int и сдвигается
влево на 8 битов
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 1 1
0 0 0 0 0 0 0 0
bits << 31 // сдвиг влево на 31 бит отбрасывает
крайние левые биты
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
bits >> 3 // сдвиг вправо на 3 бита отбрасывает 3
крайних правых бита
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 1 1
```

Побитовый оператор NOT

Побитовый оператор NOT (`~`) (bitwise NOT operator) создает новое значение с инвертированными битами своего операнда. Каждый бит, содержащий 1, превращается в 0; каждый бит, содержащий 0, — в 1.

```
unsigned char bits = 0227; 1 0 0 1 0 1 1 1
~bits
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 1 1 0 1 0 0 0
```

Здесь operand типа `char` сначала преобразуется в тип `int`. Это оставляет значение неизменным, но добавляет нулевые биты в позиции

старших разрядов. Таким образом, преобразование в тип `int` добавляет 24 бита старших разрядов, заполненных нулями. Биты преобразованного значения инвертируются.

Побитовые операторы AND, OR и XOR

Побитовые операторы `AND` (`&`), `OR` (`|`) и `XOR` (`^`) создают новые значения с битовым шаблоном, состоящим из двух их operandов.

```
unsigned char b1 = 0145; 0 1 1 0 0 1 0 1  
unsigned char b2 = 0257; 1 0 1 0 1 1 1 1  
b1 & b2 Все 24 старших бита 0 0 0 1 0 0 1 0 1  
b1 | b2 Все 24 старших бита 0 1 1 1 0 1 1 1 1  
b1 ^ b2 Все 24 старших бита 0 1 1 0 0 1 0 1 0
```

Каждая битовая позиция результата *побитового оператора AND* (`&`) содержит 1, если оба операнда содержат 1 в этой позиции; в противном случае результат — 0. У *побитового оператора OR* (`|`) бит содержит 1, если один или оба операнда содержат 1; в противном случае результат — 0. Для *побитового оператора XOR* (`^`) бит содержит 1, если любой, но не оба операнда содержат 1; в противном случае результат — 0.



ВНИМАНИЕ

Побитовые и логические (см. раздел 4.3) операторы нередко путают. Например, путают побитовый оператор `&` с логическим `&&`, побитовый `|` с логическим `||` и побитовый `~` с логическим `!`.

Использование побитовых операторов

Рассмотрим пример использования побитовых операторов. Предположим, что есть класс с 30 учениками. Каждую неделю класс отвечает на контрольные вопросы с оценкой "сдано/не сдано". Результаты всех контрольных записываются, по одному биту на ученика, чтобы представить успешную оценку или нет. Каждую контрольную можно представить в виде беззнакового целочисленного значения.

```
unsigned long quiz1 = 0; // это значение  
используется  
// как коллекция битов
```

Переменная `quiz1` определена как `unsigned long`. Таким образом, на любой машине она будет содержать по крайней мере 32 бита. Переменная `quiz1` инициализируется явно, чтобы ее значение было

определенено изначально.

Учитель должен быть способен устанавливать и проверять отдельные биты. Например, должна быть возможность установить бит, соответствующий ученику номер 27, означающий, что этот ученик сдал контрольную. Чтобы указать, что ученик 27 прошел контрольную, создадим значение, у которого установлен только бит номер 27. Если затем применить побитовый оператор OR к этому значению и значению переменной `quiz1`, то все биты, кроме бита 27, останутся неизменными.

В данном примере счет битов переменной `quiz1` начинается с 0, соответствующего младшему биту, 1 соответствует следующему биту и т.д.

Чтобы получить значение, означающее, что ученик 27 сдал контрольную, используется оператор сдвига влево и целочисленный литерал 1 типа `unsigned long` (см. раздел 2.1.3).

```
1UL << 27 // создает значение только с одним  
установленным битом  
// в позиции 27
```

Первоначально переменная `1UL` имеет 1 в самом младшем бите и по крайней мере 31 нулевой бит. Она определена как `unsigned long`, поскольку тип `int` гарантированно имеет только 16 битов, а необходимо по крайней мере 27. Это выражение сдвигает 1 на 27 битовых позиций, вставляя в биты позади 0.

К этому значению и значению переменной `quiz1` применяется оператор OR. Поскольку необходимо изменить значение самой переменной `quiz1`, используем составной оператор присвоения (см. раздел 4.4):

```
quiz1 |= 1UL << 27; // указать, что ученик номер 27  
сдал контрольную
```

Оператор `|=` выполняется аналогично оператору `+=`.

```
quiz1 = quiz1 | 1UL << 27; // эквивалент quiz1 |=  
1UL << 21;
```

Предположим, что учитель пересмотрел контрольные и обнаружил, что ученик 27 фактически списал работу. Теперь учитель должен сбросить бит 27 в 0. На сей раз необходимо целое число, бит 27 которого сброшен, а все остальные установлены в 1. Применение побитового AND к этому значению и значению переменной `quiz1` позволяет сбросить только данный бит:

```
quiz1 &= ~(1UL << 27); // ученик номер 27 не прошел  
контрольную
```

Мы получаем значение со всеми установленными битами, кроме бита 27, инвертируя предыдущее значение. У него все биты были сброшены в 0, кроме бита 27, который был установлен в 1. Применение побитового NOT к этому значению сбросит бит 27, а все другие установит. Применение побитового AND к этому значению и значению переменной `quiz1` оставит неизменными все биты, кроме бита 27.

И наконец, можно узнать, как дела у ученика 27:

```
bool status = quiz1 & (1UL << 27); // как дела у  
ученика 27?
```

Здесь оператор AND применяется к значению с установленным битом 27 и значением переменной `quiz1`. Результат отличен от нуля (т.е. истинен), если бит 27 в значении переменной `quiz1` установлен; в противном случае он нулевой.



Операторы сдвига (они же ввода и вывода) имеют левосторонний порядок

Хотя многие программисты никогда не используют побитовые операторы непосредственно, почти все они использует их перегруженные версии в виде операторов ввода и вывода. Перегруженный оператор имеет тот же приоритет и порядок, что и егостроенная версия. Поэтому программисты должны иметь понятие о приоритете и порядке операторов сдвига, даже если они никогда не используют ихстроенные версии.

Поскольку операторы сдвига имеют левосторонний порядок, выражение

```
cout << "hi" << " there" << endl;
```

выполняется так:

```
( ( cout << "hi" ) << " there" ) << endl;
```

В этом операторе operand "hi" группируется с первым символом `<<`. Его результат группируется со вторым, а его результат с третьим символом.

Приоритет операторов сдвига средний: ниже, чем у арифметических операторов, но выше, чем у операторов отношения, присвоения и условных операторов. Эти различия в уровнях приоритета свидетельствуют о том, что для правильной группировки операторов с более низким приоритетом следует использовать круглые скобки.

```
cout << 42 + 10; // ok: приоритет + выше, поэтому  
выводится сумма
```

```
cout << (10 < 42); // ok: группировку определяют скобки; выводится 1
cout << 10 < 42; // ошибка: попытка сравнить cout с 42!
```

Последний оператор `cout` интерпретируется так
`(cout << 10) < 42;`

Он гласит: "записать 10 в поток `cout`, а затем сравнить результат (т.е. поток `cout`) со значением 42".

Упражнения раздела 4.8

Упражнение 4.25. Каково значение выражения `~'q' << 6` на машине с 32-битовыми целыми числами и 8-битовыми символами, с учетом, что символ '`q`' имеет битовое представление `01110001`?

Упражнение 4.26. Что будет, если в приведенном выше примере оценки учеников использовать для переменной `quiz1` тип `unsigned int`?

Упражнение 4.27. Каков результат каждого из этих выражений?

```
unsigned long ull = 3, ul2 = 7;
(a) ull & ul2  (b) ull | ul2
(c) ull && ul2 (d) ull || ul2
```

4.9. Оператор `sizeof`

Оператор `sizeof` возвращает размер в байтах результата выражения или указанного по имени типа. Оператор имеет правосторонний порядок. Результат оператора `sizeof` — это константное выражение (см. раздел 2.4.4) типа `size_t` (см. раздел 3.5.2). Оператор существует в двух формах.

```
sizeof( тип)
sizeof выражение
```

Во второй форме оператор `sizeof` возвращает размер типа, возвращаемого выражением. Оператор `sizeof` необычен тем, что он не выполняет свой операнд.

```
Sales_data data, *p;
sizeof(Sales_data); // размер, необходимый для хранения объекта
                     // типа Sales_item
sizeof data;        // размер типа данных, аналог
sizeof(Sales_data)
sizeof p;           // размер указателя
```

```

        sizeof *p;                                // размер типа, на который
указывает указатель p,
                                                // т.е. sizeof(Sales_data)
        sizeof data.revenue; // размер типа члена revenue
класса Sales_data
        sizeof Sales_data::revenue; // альтернативный
способ получения
                                                // размера revenue

```

Наиболее интересен пример `sizeof *p`. Во-первых, поскольку оператор `sizeof` имеет правосторонний порядок и тот же приоритет, что и оператор `*`, это выражение группируется справа налево. Таким образом, оно эквивалентно выражению `sizeof(*p)`. Во-вторых, поскольку оператор `sizeof` не выполняет свой операнд, не имеет значения, допустим ли указатель `p` (т.е. инициализирован ли он) (см. раздел 2.3.2). Обращения к значению недопустимого указателя оператор `sizeof` не осуществляет, и указатель фактически не используется, поэтому он безопасен. Ему и не нужно обращаться к значению указателя, чтобы выяснить, какой тип он возвратит.



По новому стандарту для доступа к члену класса при получении его размера можно использовать оператор области видимости. Обычно к членам класса можно обратиться только через объект этого класса. Больше не обязательно предоставлять объект, так как оператор `sizeof` не обязан выбирать член класса, чтобы узнать его размер.

Результат применения оператора `sizeof` частично зависит от типа, к которому он применен.

- Если это тип `char` или выражения, результат которого имеет тип `char`, то это гарантированно будет 1.
- Если это ссылка, то возвращает размер типа объекта, на который она ссылается.
- Если это указатель, то возвращается размер, необходимый для хранения указателя.
- Если это обращение к значению указателя, то возвращается размер типа объекта, на который он указывает, вне зависимости от его допустимости.
- Если это массив, то возвращается размер всего массива. Это эквивалентно получению размера элемента массива и его умножению на

количество элементов. Обратите внимание, что оператор `sizeof` не преобразует массив в указатель.

- Если это строка или вектор, то возвращается размер только фиксированной части этих типов; но не размер, используемый элементами объекта.

Поскольку оператор `sizeof` возвращает размер всего массива, разделив размер массива на размер элемента, можно определить количество элементов в массиве:

```
// sizeof(ia)/sizeof(*ia) возвращает количество
элементов в ia
constexpr size_t sz = sizeof(ia)/sizeof(*ia);
int arr2[sz]; // ok: sizeof возвращает константное
выражение
// (p. 2.4.4)
```

Так как оператор `sizeof` возвращает константное выражение, его результат можно использовать в выражении для определения размерности массива.

Упражнения раздела 4.9

Упражнение 4.28. Напишите программу для вывода размера каждого из встроенных типов.

Упражнение 4.29. Предскажите вывод следующего кода и объясните свое рассуждение. Напишите и выполните соответствующую программу. Совпадает ли вывод с ожиданиями? Если нет, то объясните почему.

```
int x[10]; int *p = x;
cout << sizeof(x)/sizeof(*x) << endl;
cout << sizeof(p)/sizeof(*p) << endl;
```

Упражнение 4.30. Используя таблицу из раздела 4.12, расставьте скобки в следующих выражениях так, чтобы продемонстрировать порядок его обработки:

- (a) `sizeof x + y` (b) `sizeof p->mem[i]`
- (c) `sizeof a < b` (d) `sizeof f()`

4.10. Оператор запятая

Оператор запятая (,) (comma operator) получает два операнда, обрабатываемых слева направо. Подобно операторам логического AND и OR, а также условному оператору, оператор запятая гарантирует порядок обработки своих operandов.

Левое выражение обрабатывается, а его результат отбрасывается. Результат выражения запятая — это значение правого выражения. Результат является l-значением, если правый операнд — l-значение.

Оператор запятая нередко используется в цикле `for`:

```
vector<int>::size_type cnt = ivec.size();
// присвоить значения элементам size... 1 вектора
ivec
for (vector<int>::size_type ix = 0;
     ix != ivec.size(); ++ix, --cnt)
    ivec[ix] = cnt;
```

Здесь выражения в заголовке цикла `for` увеличивают значение итератора `ix` и уменьшают значение целочисленной переменной `cnt`. Значения итератора `ix` и переменной `cnt` изменяются при каждой итерации цикла. Пока проверка итератора `ix` проходит успешно, следующему элементу присваивается текущее значение переменной `cnt`.

Упражнения раздела 4.10

Упражнение 4.31. Программа этого раздела использовала префиксные операторы инкремента и декремента. Объясните, почему были использованы префиксные, а не постфиксные версии? Что следует изменить для использования постфиксных версий? Перепишите программу с использованием постфиксных операторов.

Упражнение 4.32. Объясните следующий цикл:

```
constexpr int size = 5;
int ia[size] = {1, 2, 3, 4, 5};
for (int *ptr = ia, ix = 0;
     ix != size && ptr != ia+size; ++ix, ++ptr) {
/* ... */ }
```

Упражнение 4.33. Используя таблицу раздела 4.12, объясните, что делает следующее выражение:

```
someValue ? ++x, ++y : --x, --y
```



4.11. Преобразование типов

В языке C++ некоторые типы взаимосвязаны. Когда два типа взаимосвязаны, объект или значение одного типа можно использовать там, где ожидается операнд связанного типа. Два типа считаются связанными, если между ними возможно *преобразование* (conversion).

Для примера рассмотрим следующее выражение, инициализирующее переменную `ival` значением 6:

```
int ival = 3.541 + 3; // компилятор может предупредить о потере точности
```

Операндами сложения являются значения двух разных типов: 3.541 имеет тип `double` а 3 — `int`. Вместо попытки суммирования двух значений разных типов язык C++ определяет набор преобразований, позволяющих преобразовать операнды в общий тип. Эти преобразования выполняются автоматически без вмешательства программиста, а иногда и без его ведома. Поэтому они и называются *неявным преобразованием* (*implicit conversion*).

Неявные преобразования между арифметическими типами определены так, чтобы по возможности сохранять точность. Как правило, если у выражения есть и целочисленное значение, и значение с плавающей запятой, целое число преобразуется в число с плавающей точкой. В данном случае значение 3 преобразуется в тип `double`, осуществляется сложение чисел с плавающей запятой, и возвращается результат типа `double`.

Затем происходит инициализация. При инициализации доминирует тип инициализируемого объекта. Поэтому инициализатор преобразуется в его тип. В данном случае результат сложения типа `double` преобразуется в тип `int` и используется для инициализации переменной `ival`. Преобразование типа `double` в тип `int` усекает значение типа `double`, отбрасывая десятичную часть. В данном случае выражение присваивает переменной `ival` значение 6.

Когда происходят неявные преобразования

Компилятор автоматически преобразует операнды при следующих обстоятельствах.

- В большинстве выражений значения целочисленных типов, меньших,

чем `int`, сначала преобразуются в соответствующий больший целочисленный тип.

- В условиях нелогические выражения преобразуются в тип `bool`.
- При инициализации инициализатор преобразуется в тип переменной; при присвоении правый операнд преобразуется в тип левого.
- В арифметических выражениях и выражениях отношения с операндами смешанных типов происходит преобразование в общий тип.
- Преобразования происходят также при вызове функций, как будет продемонстрировано в главе 6.



4.11.1. Арифметические преобразования

Арифметические преобразования (arithmetic conversion), впервые представленные в разделе 2.1.2, преобразуют один арифметический тип в другой. Иерархию преобразований типов определяют правила, согласно которым операнды операторов преобразуются в самый большой общий тип. Например, если один operand имеет тип `long double`, то второй operand преобразуется тоже в тип `long double` независимо от своего типа. Короче говоря, в выражениях, где используются целочисленные значения и значения с плавающей точкой, целочисленное значение преобразуется в соответствующий тип с плавающей точкой.

Целочисленные преобразования

Целочисленное преобразование (integral promotion) преобразовывает значения малых целочисленных типов в большие. Типы `bool`, `char`, `signed char`, `unsigned char`, `short` и `unsigned short` преобразуются в `int`, если значение соответствует ему, а в противном случае оно преобразуется в тип `unsigned int`. Как уже неоднократно упоминалось, значение `false` типа `bool` преобразуется в 0, а `true` в 1.

Большие символьные типы (`wchar_t`, `char16_t` и `char32_t`) преобразуются в наименьший целочисленный тип `int`, `unsigned int`, `long`, `unsigned long`, `long long` или `unsigned long long`, которому соответствуют все возможные значения этого символьного типа.

Операнды беззнакового типа

Если operandы оператора имеют разные типы, они обычно

преобразуются в общий тип. Если любой из операндов имеет беззнаковый тип, то тип, в который преобразуются операнды, зависит от относительных размеров целочисленных типов на машине.

Как обычно, сначала осуществляются целочисленные преобразования. Если полученные в результате типы совпадают, то никаких дальнейших преобразований не нужно. Если оба (возможно преобразованных) операнда имеют одинаковый знак, то операнд с меньшим типом преобразуется в больший тип.

При разных знаках, если тип беззнакового операнда больший, чем у знакового операнда, знаковый операнд преобразуется в беззнаковый. Например, при операторах типа `unsigned int` и `int`, `int` преобразуется в `unsigned int`. Следует заметить, что если значение типа `int` отрицательное, результат преобразуется так, как описано в разделе 2.1.2.

Остается случай, когда тип знакового операнда больше, чем беззнакового. В данном случае результат зависит от конкретной машины. Если все значения беззнакового типа соответствуют большему типу, то операнд беззнакового типа преобразуется в знаковый. Если значения не соответствуют, то знаковый операнд преобразуется в беззнаковый. Например, если операнды имеют типы `long` и `unsigned int` и размер у них одинаковый, операнд типа `long` будет преобразован в `unsigned int`. Если тип `long` окажется больше, то `unsigned int` будет преобразован в `long`.

Концепция арифметических преобразований

Арифметические преобразования проще всего изучить на примерах.

```
bool flag; char cval;
short sval; unsigned short usval;
int ival; unsigned int uival;
long lval; unsigned long ulval;
float fval; double dval;
3.14159L + 'a'; // 'a' преобразуется в int, а затем
int в long double
dval + ival; // ival преобразуется в double
dval + fval; // fval преобразуется в double
ival = dval; // dval преобразуется в int (с
усечением)
flag = dval; // если dval - 0, flag - false, в
противном случае - true
```

```

cval + fval; // cval преобразуется в int, затем
int во float
sval + cval; // sval и cval преобразуются в int
cval + lval; // cval преобразуется в long
ival + ulval; // ival преобразуется в unsigned long
usval + ival; // преобразование зависит от
соотношения
// размеров типов unsigned short и
int
uival + lval; // преобразование зависит от
соотношения
// размеров типов unsigned int и long

```

В первом выражении суммы символьная константа 'а' имеет тип `char`, являющийся числовым (см. раздел 2.1.1). Какое именно это значение, зависит от используемого машиной набора символов. На машине авторов, где установлен набор символов ASCII, символу 'а' соответствует число 97. При добавлении символа 'а' к значению типа `long double` значение типа `char` преобразуется в тип `int`, а затем в тип `long double`. Это преобразованное значение добавляется к литералу. Интересны также два последних случая, где происходит преобразование беззнаковых значений. Тип результата этих выражений зависит от конкретной машины.

Упражнения раздела 4.11.1

Упражнение 4.34. С учетом определений переменных данного раздела объясните, какие преобразования имеют место в следующих выражениях:

- (a) `if (fval)`
- (b) `dval = fval + ival;`
- (c) `dval + ival * cval;`

Помните, что возможно придется учитывать порядок операторов.

Упражнение 4.35. С учетом определений

```

char cval; int ival; unsigned int ui;
float fval; double dval;

```

укажите неявные преобразования типов, если такие вообще имеются.

- (a) `cval = 'а' + 3;`
- (b) `fval = ui - ival * 1.0;`
- (c) `dval = ui * fval;`
- (d) `cval = ival + fval + dval;`



4.11.2. Другие неявные преобразования

Кроме арифметических, существует еще несколько видов неявных преобразований, включая следующие.

Преобразование массива в указатель. В большинстве выражений, когда используется массив, он автоматически преобразуется в указатель на свой первый элемент.

```
int ia[10]; // массив из десяти целых чисел
int* ip = ia; // ia преобразуется в указатель на
первый элемент
```

Это преобразование не происходит при использовании массива в выражении `decltype` или в качестве операнда операторов обращения к адресу (`&`), `sizeof` или `typeid` (который рассматривается в разделе 19.2.2). Преобразование не происходит также при инициализации ссылки на массив (см. раздел 3.5.1). Подобное преобразование указателя происходит при использовании в выражении типа функции, как будет описано в разделе 6.7.

Преобразование указателя. Существует несколько других преобразований указателя: постоянное целочисленное значение 0 и литерал `nullptr` могут быть преобразованы в указатель на любой тип; указатель на любой неконстантный тип может быть преобразован в `void*`, а указатель на любой тип может быть преобразован в `const void*`. Как будет продемонстрировано в разделе 15.2.2, существуют дополнительные преобразования указателя, относящиеся к типам, связанным наследованием.

Преобразование в тип `bool`. Существует автоматическое преобразование арифметических типов и типов указателя в тип `bool`. Если указатель или арифметическое значение — нуль, преобразование возвращает значение `false`; любое другое значение возвращает `true`:

```
char *cp = get_string();
if (cp) /* ... */ // true, если cp не нулевой
указатель
while (*cp) /* ... */ // true, если *cp не нулевой
символ
```

Преобразование в константу. Указатель на неконстантный тип можно преобразовать в указатель на соответствующий константный тип, то же относится и к ссылкам. Таким образом, если `T` — тип, то указатель или ссылку на тип `T` можно преобразовать в указатель или ссылку на `const T`.

(см. разделы 2.4.1 и 2.4.2).

```
int i;
const int &j = i;      // преобразовать в ссылку на
const int
const int *p = &i;    // преобразовать неконстантный
адрес в константный
int &r = j, *q = p;   // ошибка: преобразование
константы в не константу
                           // недопустимо
```

Обратное преобразование (удаление спецификатора `const` нижнего уровня) невозможно.

Преобразование, определенное типами класса. Тип класса может сам определять преобразования, которые компилятор применит автоматически. Компилятор применяет только одно преобразование типа класса за раз. В разделе 7.5.4 приведен пример, когда необходимо несколько преобразований, и он не работает.

В программах ранее уже использовались преобразования типов класса, когда символьная строка в стиле С использовалась там, где ожидался библиотечный тип `string` (см. раздел 3.5.5), а также при чтении из потока `istream` в условии.

```
string s, t = "a value"; // символьный строковый
литерал преобразован
                           // в тип string
while (cin >> s)           // условие while
преобразует cin в bool
```

Условие (`cin >> s`) читает поток `cin` и возвращает его же как результат. Условия ожидают значение типа `bool`, но оно проверяет значение типа `istream`. Библиотека IO определяет преобразование из типа `istream` в `bool`. Это преобразование используется автоматически, чтобы преобразовать поток `cin` в тип `bool`. Результатирующее значение типа `bool` зависит от состояния потока. Если последнее чтение успешно, то преобразование возвращает значение `true`. Если последняя попытка потерпела неудачу, то преобразование возвращает значение `false`.

4.11.3. Явные преобразования

Иногда необходимо явно преобразовать объект в другой тип. Например, в следующем коде может понадобиться использование деления с плавающей точкой:

```
int i, j;  
double slope = i/j;
```

Для этого необходим способ явного преобразования переменных *i* и/или *j* в тип *double*. Для явного преобразования используется *приведение* (*cast*) типов.



Хотя приведение время от времени необходимо, оно довольно опасно.

Именованные операторы приведения

Именованный оператор приведения имеет следующую форму:

имя_приведения<тип>(выражение) ;

где *тип* — это результирующий тип преобразования, а *выражение* — приводимое значение. Если *тип* — ссылка, то результат l-значение. *Имя_приведения* может быть одним из следующих: *static_cast*, *dynamic_cast*, *const_cast* и *reinterpret_cast*. Приведение *dynamic_cast*, обеспечивающее идентификацию типов времени выполнения, рассматривается в разделе 19.2. *Имя_приведения* определяет, какое преобразование осуществляется.

Оператор static_cast

Любое стандартное преобразование типов, кроме задействующего спецификатор *const* нижнего уровня, можно затребовать, используя оператор *static_cast*. Например, приведя тип одного из operandов к типу *double*, можно заставить выражение использовать деление с плавающей точкой:

```
// приведение для вынужденного деления с плавающей  
точкой
```

```
double slope = static_cast<double>(j) / i;
```

Оператор *static_cast* зачастую полезен при присвоении значения большего арифметического типа переменной меньшего. Приведение сообщает и читателю программы, и компилятору, что мы знаем и не беспокоимся о возможной потере точности. При присвоении большего арифметического типа меньшему компиляторы зачастую выдают предупреждение. При явном приведении предупреждающее сообщение не выдается.

Оператор `static_cast` полезен также при выполнении преобразований, которые компилятор не выполняет автоматически. Например, его можно использовать для получения значения указателя, сохраняемого в указателе `void*` (см. раздел 2.3.2):

```
void* p = &d; // ok: адрес любого неконстантного объекта может
```

```
                                // храниться в указателе void*
// ok: преобразование void* назад в исходный тип
указателя
```

```
double *dp = static_cast<double*>(p);
```

После сохранения адреса в указателе типа `void*` можно впоследствии использовать оператор `static_cast` и привести указатель к его исходному типу, что позволит сохранить значение указателя. Таким образом, результат приведения будет равен первоначальному значению адреса. Однако следует быть абсолютно уверенным в том, что тип, к которому приводится указатель, является фактическим типом этого указателя; при несоответствии типов результат непредсказуем.

Оператор `const_cast`

Оператор `const_cast` изменяет только спецификатор `const` нижнего уровня своего операнда (см. раздел 2.4.3):

```
const char *pc;
char *p = const_cast<char*>(pc); // ok: однако
запись при помощи p
                                // указателя
```

непредсказуема

Принято говорить, что приведение, преобразующее константный объект в неконстантный, "сбрасывает `const`". При сбросе константности объекта компилятор больше не будет препятствовать записи в этот объект. Если объект первоначально не был константным, использование приведения для доступа на запись вполне допустимо. Но применение оператора `const_cast` для записи в первоначально константный объект непредсказуемо.

Только оператор `const_cast` позволяет изменить константность выражения. Попытка изменить константность выражения при помощи любого другого именованного оператора приведения закончится ошибкой компиляции. Аналогично нельзя использовать оператор `const_cast` для изменения типа выражения:

```

const char *cp;
// ошибка: static_cast не может сбросить const
char *q = static_cast<char*>( cp );
static_cast<string>( cp ); // ok: преобразует
строковый литерал в строку
const_cast<string>( cp ); // ошибка: const_cast
изменяет только
// константность

```

Оператор `const_cast` особенно полезен в контексте перегруженных функций, рассматриваемых в разделе 6.4.

Оператор `reinterpret_cast`

Оператор `reinterpret_cast` осуществляет низкоуровневую интерпретацию битовой схемы своих operandов. Рассмотрим, например, следующее приведение:

```

int *ip;
char *pc = reinterpret_cast<char*>( ip );

```

Никогда не следует забывать, что фактическим объектом, на который указывает указатель `pc`, является целое число, а не символ. Любое использование указателя `pc`, подразумевающее, что это обычный символьный указатель, вероятно, потерпит неудачу во время выполнения. Например, следующий код, вероятней всего, приведет к непредвиденному поведению во время выполнения:

```
string str( pc );
```

Использование указателя `pc` для инициализации объекта типа `string` — хороший пример небезопасности оператора `reinterpret_cast`. Проблема в том, что при изменении типа компилятор не выдаст никаких предупреждений или сообщений об ошибке. При инициализации указателя `pc` адресом типа `int` компилятор не выдаст ни предупреждения, ни сообщения об ошибке, поскольку явно указано, что это и нужно. Однако любое последующее применение указателя `pc` подразумевает, что он содержит адрес значения типа `char*`. Компилятор не способен выяснить, что фактически это указатель на тип `int`. Таким образом, инициализация строки `str` при помощи указателя `pc` вполне правомерна, хотя в данном случае абсолютно бессмысленна, если не хуже! Отследить причину такой проблемы иногда чрезвычайно трудно, особенно если приведение указателя `ip` к `pc` происходит в одном файле, а использование указателя `pc` для инициализации объекта класса `string` — в другом.



Оператор `reinterpret_cast` жестко зависит от конкретной машины. Чтобы безопасно использовать оператор `reinterpret_cast`, следует хорошо понимать, как именно реализованы используемые типы, а также то, как компилятор осуществляет приведение.

Приведение типов в старом стиле

В ранних версиях языка C++ явное приведение имело одну из следующих двух форм:

```
тип ( выражение ); // форма записи приведения в стиле
функции
```

```
( тип ) выражение; // форма записи приведения в стиле
языка С
```

В зависимости от используемых типов, приведение старого стиля срабатывает аналогично операторам `const_cast`, `static_cast` или `reinterpret_cast`. В случаях, где используются операторы `static_cast` или `const_cast`, приведение типов в старом стиле позволяет осуществить аналогичное преобразование, что и соответствующий именованный оператор приведения. Но если ни один из подходов не допустим, то приведение старого стиля срабатывает аналогично оператору `reinterpret_cast`. Например, используя форму записи старого стиля, можно получить тот же результат, что и с использованием `reinterpret_cast`.

```
char *pc = (char*) ip; // ip указатель на тип int
```

Совет. Избегайте приведения типов

Приведение нарушает обычный порядок контроля соответствия типов (см. раздел 2.2), поэтому авторы настоятельно рекомендуют избегать приведения типов. Это особенно справедливо для оператора `reinterpret_cast`. Такие приведения всегда опасны. Операторы `const_cast` могут быть весьма полезны в контексте перегруженных функций, рассматриваемых в разделе 6.4. Использование оператора `const_cast` зачастую свидетельствует о плохом проекте. Другие операторы приведения, `static_cast` и `dynamic_cast`, должны быть необходимы нечасто. При каждом применении приведения имеет смысл хорошо подумать, а нельзя ли получить тот же результат другим

способом. Если приведение все же неизбежно, имеет смысл принять меры, позволяющие снизить вероятность возникновения ошибки, т.е. ограничить область видимости, в которой используется приведенное значение, а также хорошо документировать все подобные случаи.



Приведения старого стиля менее очевидны, чем именованные операторы приведения. Поскольку их легко упустить из виду, обнаружить ошибку становится еще трудней.

Упражнения раздела 4.11.3

Упражнение 4.36. С учетом того, что `i` имеет тип `int`, `a d` — `double`, напишите выражение `i *= d` так, чтобы осуществлялось целочисленное умножение, а не с плавающей запятой.

Упражнение 4.37. Перепишите каждое из следующих приведений старого стиля так, чтобы использовался именованный оператор приведения.

```
int i; double d; const string *ps; char *pc; void
*pv;
(a) pv = (void*) ps; (b) i = int(*pc);
(c) pv = &d; (d) pc = (char*) pv;
```

Упражнение 4.38. Объясните следующее выражение:
`double slope = static_cast<double>(j/i);`

4.12. Таблица приоритетов операторов

Таблица 4.4. Приоритет операторов

Порядок и оператор	Действие	Применение	Раздел
Л <code>::</code>	Глобальная область видимости	<code>:: имя</code>	7.4.1
Л <code>::</code>	Область видимости класса	<code>класс:: имя</code>	3.2.2
Л <code>::</code>	Область видимости пространства имен	<code>пространствоимен:: имя</code>	3.1
Л <code>.</code>	Обращение к члену класса	<code>объект. член</code>	1.5.2
Л <code>-></code>	Обращение к члену класса	<code>pointer->член</code>	3.4.1

$\Lambda []$	Индексирование	$выражение[выражение]$	3.5.2	
$\Lambda ()$	Вызов функции	$имя(список_выражений)$	1.5.2	
$\Lambda ()$	Конструкция <code>typeid</code>	$тип(список_выражений)$	4.11.3	
$\Lambda ++$	Постфиксный инкремент	$l\text{-значение}++$	4.5	
$\Lambda --$	Постфиксный декремент	$l\text{-значение}--$	4.5	
$\Lambda typeid$	Идентификатор типа	$typeid(тип)$	19.2.2	
$\Lambda typeid$	Идентификатор типа времени выполнения	$typeid(выражение)$	19.2.2	
Λ Явное приведение	Преобразование типов	$cast_имя<тип>(выражение)$	4.11.3	
$\Lambda ++$	Префиксный инкремент	$++l\text{-значение}$	4.5	
$\Lambda --$	Префиксный декремент	$--l\text{-значение}$	4.5	
$\Lambda \sim$	Побитовое NOT	$\sim выражение$	4.8	
$\Lambda !$	Логическое NOT	$! выражение$	4.3	
$\Lambda -$	Унарный минус	$- выражение$	4.2	
$\Lambda +$	Унарный плюс	$+ выражение$	4.2	
$\Lambda *$	Обращение к значению	$* выражение$	2.3.2	
$\Lambda &$	Обращение к адресу	$& l\text{-значение}$	2.3.2	
$\Lambda ()$	Преобразование типов	$(тип) выражение$	4.11.3	
$\Lambda sizeof$	Размер объекта	$sizeof выражение$	4.9	
$\Lambda sizeof$	Размер типа	$sizeof(тип)$	4.9	
$\Lambda sizeof...$	Размер пакета параметров	$sizeof... (имя)$	16.4	
Λnew	Создание объекта	$new тип$	12.1.2	
$\Lambda new[]$	Создание массива	$new тип[размер]$	12.1.2	
$\Lambda delete$	Освобождение объекта	$delete выражение$	12.1.2	
$\Lambda delete[]$	Освобождение массива	$delete[] выражение$	12.1.2	
$\Lambda noexcept$	Способность к передаче	$noexcept(выражение)$	18.1.4	
$\Lambda ->*$	Указатель на член класса	$указатель->* указатель_на_член$	19.4.1	
$\Lambda . *$	Указатель на член класса	$объект. * указатель_на_член$	19.4.1	
$\Lambda *$	Умножение	$выражение * выражение$	4.2	
$\Lambda /$	Деление	$выражение / выражение$	4.2	
$\Lambda %$	Деление по модулю (остаток)	$выражение \% выражение$	4.2	

$\Lambda +$	Сумма	выражение + выражение	4.2	
$\Lambda -$	Разница	выражение - выражение	4.2	
$\Lambda <<$	Побитовый сдвиг влево	выражение << выражение	4.8	
$\Lambda >>$	Побитовый сдвиг вправо	выражение >> выражение	4.8	
$\Lambda <$	Меньше	выражение < выражение	4.3	
$\Lambda \leq$	Меньше или равно	выражение \leq выражение	4.3	
$\Lambda >$	Больше	выражение $>$ выражение	4.3	
$\Lambda \geq$	Больше или равно	выражение \geq выражение	4.3	
$\Lambda ==$	Равенство	выражение $= =$ выражение	4.3	
$\Lambda !=$	Неравенство	выражение $!=$ выражение	4.3	
$\Lambda \&$	Побитовый AND	выражение $\&$ выражение	4.8	
$\Lambda ^$	Побитовый XOR	выражение \wedge выражение	4.8	
$\Lambda $	Побитовый OR	выражение $ $ выражение	4.8	
$\Lambda \&\&$	Логический AND	выражение $\&\&$ выражение	4.3	
$\Lambda $	Логический OR	выражение $ $ выражение	4.3	
$\Pi ?:$	Условный оператор	выражение ? выражение : выражение	4.7	
$\Pi =$	Присвоение	1-значение $=$ выражение	4.4	
$\Pi *=, /=, \%=,$	Составные операторы присвоения	1-значение $+=$ выражение, и т.д.	4.4	
$\Pi +=, -=,$			4.4	
$\Pi <<=, >>=,$			4.4	
$\Pi \&=, =, ^=$			4.4	
Πthrow	Передача исключения	throw выражение	4.6.1	
$\Lambda ,$	Запятая	выражение, выражение	4.10	

Резюме

Язык C++ предоставляет богатый набор операторов и определяет их назначение, когда они относятся к значениям встроенных типов. Кроме того, язык поддерживает перегрузку операторов, позволяющую самостоятельно определять назначение операторов для типов класса. Определение операторов для собственных типов рассматривается в главе 14.

Чтобы разобраться в составных выражениях (содержащих несколько

операторов), необходимо выяснить приоритет и порядок обработки операндов. Каждый оператор имеет приоритет и порядок. Приоритет определяет группировку операторов в составном выражении, а порядок определяет группировку операторов с одинаковым уровнем приоритета.

Для большинства операторов порядок выполнения операндов не определен, компилятор выбирает сам, какой operand обработать сначала — левый или правый. Зачастую порядок вычисления результатов операндов никак не влияет на результат выражения. Но если оба операнда обращаются к одному объекту, причем один из них *изменяет* объект, то порядок выполнения становится весьма важен, а связанные с ним серьезные ошибки обнаружить крайне сложно.

И наконец, компилятор зачастую сам преобразует тип операндов в другой связанный тип. Например, малые целочисленные типы преобразуются в *больший* целочисленный тип каждого выражения. Преобразования существуют и для встроенных типов, и для классов. Преобразования могут быть также осуществлены явно, при помощи приведения.

Термины

L-значение (l-value). Выражение, возвращающее объект или функцию. Неконстантное l-значение обозначает объект, который может быть левым операндом оператора присвоения.

R-значение (r-value). Выражение, возвращающее значение, но не ассоцииированную с ним область, если таковое значение вообще имеется.

Арифметическое преобразование (arithmetic conversion). Преобразование одного арифметического типа в другой. В контексте парных арифметических операторов арифметические преобразования, как правило, сохраняют точность, преобразуя значения меньшего типа в значения большего (например, меньший целочисленный тип `char` или `short` преобразуется в `int`).

Выражение (expression). Самый низкий уровень вычислений в программе на языке C++. Как правило, выражения состоят из одного или нескольких операторов. Каждое выражение возвращает результат. Выражения могут использоваться в качестве операндов, что позволяет создавать составные выражения, которым для вычисления собственного результата нужны результаты других выражений, являющихся его операндами.

Вычисление по сокращенной схеме (short-circuit evaluation). Термин,

описывающий способ выполнения операторов логического AND и OR. Если первого операнда этих операторов достаточно для определения общего результата, то остальные операнды не рассматриваются и не вычисляются.

Неявное преобразование (*implicit conversion*). Преобразование, которое осуществляется компилятором автоматически. Такое преобразование осуществляется в случае, когда оператор получает значение, тип которого отличается от необходимого. Компилятор автоматически преобразует operand в необходимый тип, если соответствующее преобразование определено.

Операнд (*operand*). Значение, с которым работает выражение. У каждого оператора есть один или несколько operandов

Оператор --. Оператор декремента. Имеет две формы, префиксную и постфиксную. Префиксный оператор декремента возвращает l-значение. Он вычитает единицу из значения operand и возвращает полученное значение. Постфиксный оператор декремента возвращает r-значение. Он вычитает единицу из значения operand, но возвращает исходное, неизмененное значение. Примечание: итераторы имеют оператор -- даже если у них нет оператора -.

Оператор !. Оператор логического NOT. Возвращает инверсное значение своего operand типа `bool`. Результат `true`, если operand `false`, и наоборот.

Оператор &. Побитовый оператор AND. Создает новое целочисленное значение, в котором каждая битовая позиция имеет значение 1, если оба operand в этой позиции имеют значение 1. В противном случае бит получает значение 0.

Оператор &&. Оператор логического AND. Возвращает значение `true`, если оба operand истинны. Правый operand обрабатывается, *только если* левый operand истинен.

Оператор ,. Оператор запятая. Бинарный оператор, обрабатывающийся слева направо. Результатом оператора запятая является значение справа. Результат является l-значением, только если его operand — l-значение.

Оператор ?:. Условный оператор. Сокращенная форма конструкции `if...else` следующего вида: `условие ? выражение1 : выражение2`. Если `условие` истинно (значение `true`) выполняется `выражение1`, в противном случае — `выражение2`. Тип выражений должен совпадать или допускать преобразование в общий тип.

Выполняется только одно из выражений.

Оператор ^. Побитовый оператор XOR. Создает новое целочисленное значение, в котором каждая битовая позиция имеет значение 1 , если любой (но не оба) из операндов содержит значение 1 в этой битовой позиции. В противном случае бит получает значение 0.

Оператор | . Побитовый оператор OR. Создает новое целочисленное значение, в котором каждая битовая позиция имеет значение 1 , если любой из операндов содержит значение 1 в этой битовой позиции. В противном случае бит получает значение 0 .

Оператор || . Оператор логического OR. Возвращает значение true , если любой из операндов истинен. Правый operand обрабатывается, *только если* левый operand ложен.

Оператор ~. Побитовый оператор NOT. Инвертирует биты своего operand'a.

Оператор ++. Оператор инкремента. Оператор инкремента имеет две формы, префиксную и постфиксную. Префиксный оператор инкремента возвращает l-значение. Он добавляет единицу к значению operand'a и возвращает полученное значение. Постфиксный оператор инкремента возвращает r-значение. Он добавляет единицу к значению operand'a, но возвращает исходное, неизмененное значение. Примечание: итераторы имеют оператор ++ , даже если у них нет оператора + .

Оператор <<. Оператор сдвига влево. Сдвигает биты левого operand'a влево. Количество позиций, на которое осуществляется сдвиг, задает правый operand. Правый operand должен быть нулем или положительным значением, ни в коем случае не превосходящим количества битов в левом operand'e. Левый operand должен быть беззнаковым; если левый operand будет иметь знаковый тип, то сдвиг бита знака приведет к непредсказуемому результату.

Оператор >>. Оператор сдвига вправо. Аналогичен оператору сдвига влево, за исключением направления перемещения битов. Правый operand должен быть нулем или положительным значением, ни в коем случае не превосходящим количества битов в левом operand'e. Левый operand должен быть беззнаковым; если левый operand будет иметь знаковый тип, то сдвиг бита знака приведет к непредсказуемому результату.

Оператор const_cast. Применяется при преобразовании объекта со спецификатором const нижнего уровня в соответствующий неконстантный тип, и наоборот.

Оператор dynamic_cast. Используется в комбинации с

наследованием и идентификацией типов во время выполнения. См. раздел 19.2.

Оператор `reinterpret_cast`. Интерпретирует содержимое операнда как другой тип. Очень опасен и жестко зависит от машины.

Оператор `sizeof`. Возвращает размер в байтах объекта, указанного по имени типа, или типа переданного выражения.

Оператор `static_cast`. Запрос на явное преобразование типов, которое компилятор осуществил бы неявно. Зачастую используется для переопределения неявного преобразования, которое в противном случае выполнил бы компилятор.

Оператор (`operator`). Символ, который определяет действие, выполняемое выражением. В языке определен целый набор операторов, которые применяются для значений встроенных типов. В языке определен также приоритет и порядок выполнения для каждого оператора, а также задано количество operandов для каждого из них. Операторы могут быть перегружены и применены к объектам классов.

Парный оператор (`binary operator`). Операторы, в которых используются два операнда.

Перегруженный оператор (`overloaded operator`). Версия оператора, определенного для использования с объектом класса. Определение перегруженных версий операторов рассматривается в главе 14.

Порядок (`associativity`). Определяет последовательность выполнения операторов одинакового приоритета. Операторы могут иметь правосторонний (справа налево) или левосторонний (слева направо) порядок выполнения.

Порядок вычисления (`order of evaluation`). Порядок, если он есть, определяет последовательность вычисления operandов оператора. В большинстве случаев компилятор C++ самостоятельно определяет порядок вычисления operandов. Однако, прежде чем выполнится сам оператор, всегда вычисляются его operandы. Только операторы `&&`, `||`, `?:` и `,` определяют порядок выполнения своих operandов.

Преобразование (`conversion`). Процесс, в ходе которого значение одного типа преобразуется в значение другого типа. Преобразования между встроенными типами заложены в самом языке. Для классов также возможны преобразования типов.

Преобразование (`promotion`). См. целочисленное преобразование.

Приведение (`cast`). Явное преобразование типов.

Приоритет (`precedence`). Определяет порядок выполнения операторов в

выражении. Операторы с более высоким приоритетом выполняются прежде операторов с более низким приоритетом.

Результат (result). Значение или объект, полученный при вычислении выражения.

Составное выражение (compound expression). Выражение, состоящее из нескольких операторов.

Унарный оператор (unary operator). Оператор, использующий один operand.

Целочисленное преобразование (integral promotion). Подмножество стандартных преобразований, при которых меньший целочисленный тип приводится к ближайшему большему типу. Операнды меньших целочисленных типов (например, `short`, `char` и т.д.) преобразуются всегда, даже если такие преобразования, казалось бы, необязательны.

Глава 5

Операторы

Подобно большинству языков, язык C++ предоставляет операторы для условного выполнения кода, циклы, позволяющие многократно выполнять те же фрагменты кода, и операторы перехода, прерывающие поток выполнения. В данной главе операторы, поддерживаемые языком C++, рассматриваются более подробно.

Операторы (*statement*) выполняются последовательно. За исключением самых простых программ последовательного выполнения недостаточно. Поэтому язык C++ определяет также набор операторов *управления потоком* (*flow of control*), обеспечивающих более сложные пути выполнения кода.



5.1. Простые операторы

Большинство операторов в языке C++ заканчиваются точкой с запятой. Выражение типа `ival + 5` становится *оператором выражения* (*expression statement*), завершающимся точкой с запятой. Операторы выражения составляют вычисляемую часть выражения.

```
ival + 5; // оператор выражения (хоть и  
бесполезный)
```

```
cout << ival; // оператор выражения
```

Первое выражение бесполезно: результат вычисляется, но не присваивается, а следовательно, никак не используется. Как правило, выражения содержат операторы, результат вычисления которых влияет на состояние программы. К таким операторам относятся присвоение, инкремент, ввод и вывод.

Пустые операторы

Самая простая форма оператора — это *пустой* (*empty*), или *нулевой*, *оператор* (*null statement*). Он представляет собой одиночный символ точки с запятой (`;`).

```
;// пустой оператор
```

Пустой оператор используется в случае, когда синтаксис языка требует

наличия оператора, а логика программы — нет. Как правило, это происходит в случае, когда вся работа цикла осуществляется в его условии. Например, можно организовать ввод, игнорируя все прочитанные данные, пока не встретится определенное значение:

```
// читать, пока не встретится конец файла или  
значение,  
// равное содержимому переменной sought  
while (cin >> s && s != sought)  
; // пустой оператор
```

В условии значение считывается со стандартного устройства ввода, и объект `cin` неявно проверяется на успешность чтения. Если чтение прошло успешно, во второй части условия проверяется, не равно ли полученное значение содержимому переменной `sought`. Если искомое значение найдено, цикл `while` завершается, в противном случае его условие проверяется снова, начиная с чтения следующего значения из объекта `cin`.

Рекомендуем

Случай применения пустого оператора следует комментировать, чтобы любой, кто читает код, мог сразу понять, что оператор пропущен преднамеренно.

Остерегайтесь пропущенных и лишних точек с запятой

Поскольку пустой оператор является вполне допустимым, он может располагаться везде, где ожидается оператор. Поэтому лишний символ точки с запятой, который может показаться явно недопустимым, на самом деле является не более, чем пустым оператором. Приведенный ниже фрагмент кода содержит два оператора: оператор выражения и пустой оператор.

```
ival = v1 + v2;; // ok: вторая точка с запятой –  
это лишний  
// пустой оператор
```

Хотя ненужный пустой оператор зачастую безопасен, дополнительная точка с запятой после условия цикла `while` или оператора `if` может решительно изменить поведение кода. Например, следующий цикл будет выполняться бесконечно:

```
// катастрофа: лишняя точка с запятой превратила
```

```
тело цикла
// в пустой оператор
while (iter != svec.end()) ; // тело цикла while
пусто!
++iter; // инкремент не
является частью цикла
```

Несмотря на отступ, выражение с оператором инкремента не является частью цикла. Тело цикла — это пустой оператор, обозначенный символом точки с запятой непосредственно после условия.



Лишний пустой оператор не всегда безопасен.

Составные операторы (блоки)

Составной оператор (compound statement), обычно называемый *блоком* (block), представляет собой последовательность операторов, заключенных в фигурные скобки. Блок операторов обладает собственной областью видимости (см. раздел 2.2.4). Объявленные в блоке имена доступны только в данном блоке и блоках, вложенных в него. Как обычно, имя видимо только с того момента, когда оно определено, и до конца блока включительно.

Составные операторы применяются в случае, когда язык требует одного оператора, а логика программы нескольких. Например, тело цикла `while` или `for` составляет один оператор. Но в теле цикла зачастую необходимо выполнить несколько операторов. Заключив необходимые операторы в фигурные скобки, можно получить блок, рассматриваемый как единый оператор.

Для примера вернемся к циклу `while` из кода в разделе 1.4.1.

```
while (val <= 10) {
    sum += val; // присвоить sum сумму val и sum
    ++val;      // добавить 1 к val
}
```

Логика программы нуждалась в двух операторах, но цикл `while` способен содержать только один оператор. Заключив эти операторы в фигурные скобки, получаем один (составной) оператор.



Блок *не завершают* точкой с запятой.

Как и в случае с пустым оператором, вполне можно создать пустой блок. Для этого используется пара фигурных скобок без операторов:

```
while (cin >> s && s != sought)
{ } // пустой блок
```

Упражнения раздела 5.1

Упражнение 5.1. Что такое пустой оператор? Когда его можно использовать?

Упражнение 5.2. Что такое блок? Когда его можно использовать?

Упражнение 5.3. Используя оператор запятой (см. раздел 4.10), перепишите цикл `while` из раздела 1.4.1 так, чтобы блок стал больше не нужен. Объясните, улучшило ли это удобочитаемость кода.

5.2. Операторная область видимости

Переменные можно определять в управляющих структурах операторов `if`, `switch`, `while` и `for`. Переменные, определенные в управляющей структуре, видимы только в пределах этого оператора и выходят из области видимости по его завершении.

```
while (int i = get_num()) // i создается и
инициализируется при
```

// каждой итерации

```
cout << i << endl;
i = 0; // ошибка: переменная i недоступна вне цикла
```

Если к значению управляющей переменной необходимо обращаться впоследствии, то ее следует определить вне оператора.

```
// найти первый отрицательный элемент
auto beg = v.begin();
while (beg != v.end() && *beg >= 0)
    ++beg;
if (beg == v.end())
    // известно, что все элементы v больше или равны
нулю
```

Значение объекта, определенного в управляющей структуре, используется самой структурой. Поэтому такие переменные следует инициализировать.

Упражнения раздела 5.2

Упражнение 5.4. Объясните каждый из следующих примеров, а также устраните все обнаруженные проблемы.

```
( a) while ( string::iterator iter != s.end() ) { /*  
... */ }  
( b) while ( bool status = find( word ) ) { /* ... */ }  
      if ( !status ) { /* ... */ }
```

5.3. Условные операторы

Язык C++ предоставляет два оператора, обеспечивающих условное выполнение. Оператор `if` разделяет поток выполнения на основании условия. Оператор `switch` вычисляет результат целочисленного выражения и на его основании выбирает один из нескольких путей выполнения.



5.3.1. Оператор `if`

Оператор `if` выполняет один из двух операторов в зависимости от истинности своего условия. Существуют две формы оператора `if`: с разделом `else` и без него. Синтаксис простой формы оператора `if` имеет следующий вид:

```
if (условие)
```

 оператор

Оператор `if else` имеет следующую форму:

```
if (условие)
```

 оператор

```
else
```

 оператор2

В обеих версиях `условие` заключается в круглые скобки. `Условие` может быть выражением или инициализирующим объявлением переменной (см. раздел 5.2). Тип выражения или переменной должен быть преобразуем в тип `bool` (см. раздел 4.11). Как обычно, и `оператор`, и `оператор2` могут быть блоком.

Если `условие` истинно, `оператор` выполняется. По завершении оператора выполнение продолжается после оператора `if`.

Если `условие` ложно, `оператор` пропускается. В простом операторе `if` выполнение продолжается после оператора `if`, а в операторе `if else` выполняется `оператор2`.

Использование оператора `if else`

Для иллюстрации оператора `if else` вычислим символ оценки по ее числу. Подразумевается, что числовые значения оценок находятся в диапазоне от нуля до 100 включительно. Оценка 100 получает знак "A++", оценка ниже 60 — "F", а остальные группируются по десять: от 60 до 69 — "D", от 70 до 79 — "C" и т.д. Для хранения возможных символов оценок используем вектор:

```
vector<string> scores = { "F", "D", "C", "B", "A",  
"A++" } ;
```

Для решения этой проблемы можно использовать оператор `if else`, чтобы выполнять разные действия проходных и не проходных отметок.

```

// если оценка меньше 60 - это F, в противном
случае вычислять индекс
string lettergrade;
if (grade < 60)
    lettergrade = scores[ 0 ];
else
    lettergrade = scores[ ( grade - 50 ) / 10 ];

```

В зависимости от значения переменной `grade` оператор выполняется либо после части `if`, либо после части `else`. В части `else` вычисляется индекс оценки уже без неудовлетворительных. Затем усекающее остаток целочисленное деление (см. раздел 4.2) используется для вычисления соответствующего индекса вектора `scores`.

Вложенные операторы if

Чтобы сделать программу интересней, добавим к удовлетворительным отметкам плюс или минус. Плюс присваивается оценкам, заканчивающимся на 8 или 9, а минус — заканчивающимся на 0, 1 или 2.

```

if (grade % 10 > 7)
    lettergrade += '+'; // оценки, заканчивающиеся на
8 или 9, получают +
else if (grade % 10 < 3)
    lettergrade += '-'; // оценки, заканчивающиеся на
0, 1 и 2, получают -

```

Для получения остатка и принятия на основании его решения, добавлять ли плюс или минус, используем оператор деления по модулю (см. раздел 4.2).

Теперь добавим код, присваивающий плюс или минус, к коду, выбирающему символ оценки:

```

// если оценка неудовлетворительна, нет смысла
роверять ее на + или -
if (grade < 60)
    lettergrade = scores[ 0 ];
else {
    lettergrade = scores[ ( grade - 50 ) / 10 ]; // выбрать
символ оценки
    if (grade != 100) // добавлять + или -, только если
это не A++
        if (grade % 10 > 7)
            lettergrade += '+'; // оценки, заканчивающиеся на

```

```

8 или 9,
                                // получают +
else if (grade % 10 < 3)
    lettergrade += '-'; // оценки, заканчивающиеся на
0, 1 и 2,
                                // получают -
}

```

Обратите внимание, что два оператора, следующих за первым оператором `else`, заключены в блок. Если переменная `grade` содержит значение 60 или больше, возможны два действия: выбор символа оценки из вектора `scores` и, при условии, добавление плюса или минуса.

Следите за фигурными скобками

Когда несколько операторов следует выполнить как блок, довольно часто забывают фигурные скобки. В следующем примере, вопреки отступу, код добавления плюса или минуса выполняется безусловно:

```

if (grade < 60)
    lettergrade = scores[ 0 ];
else // ошибка: отсутствует фигурная скобка
    lettergrade = scores[(grade - 50)/10];
    // несмотря на внешний вид, без фигурной скобки,
этот код
    // выполняется всегда
    // неудовлетворительным оценкам ошибочно
присваивается - или +
if (grade != 100)
    if (grade % 10 > 7)
        lettergrade += '+'; // оценки, заканчивающиеся
на 8 или 9,
                                // получают +
else if (grade % 10 < 3)
    lettergrade += '-'; // оценки, заканчивающиеся
на 0, 1 и 2,
                                // получают -

```

Найти такую ошибку бывает очень трудно, поскольку программа выглядит правильно.

Во избежание подобных проблем некоторые стили программирования рекомендуют всегда использовать фигурные скобки после оператора `if` или `else` (а также вокруг тел циклов `while` и `for`).

Это позволяет избежать подобных ошибок. Это также означает, что фигурные скобки уже есть, если последующие изменения кода потребуют добавления операторов.



Рекомендуем

У большинства редакторов и сред разработки есть инструменты автоматического выравнивания исходного кода в соответствии с его структурой. Такие инструменты всегда следует использовать, если они доступны.

Потерянный оператор else

Когда один оператор `if` вкладывается в другой, ветвей `if` может оказаться больше, чем ветвей `else`. Действительно, в нашей программе оценивания четыре оператора `if` и два оператора `else`. Возникает вопрос: как установить, которому оператору `if` принадлежит данный оператор `else`?

Эта проблема, обычно называемая *потерянным оператором else* (*dangling else*), присуща многим языкам программирования, предоставляющим операторы `if` и `if else`. Разные языки решают эту проблему по-разному. В языке C++ неоднозначность решается так: оператор `else` принадлежит ближайшему расположенному выше оператору `if` без `else`.

Неприятности происходят также, когда код содержит больше операторов `if`, чем ветвей `else`. Для иллюстрации проблемы перепишем внутренний оператор `if else`, добавляющий плюс или минус, на основании различных наборов условий:

```
// Ошибка: порядок выполнения НЕ СООТВЕТСТВУЕТ
отступам; ветвь else
// принадлежит внутреннему if
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+'; // оценки, заканчивающиеся на
8 или 9,
                                // получают +
else
    lettergrade += '-'; // оценки, заканчивающиеся на
3, 4, 5, 6,
```

// получают - !

Отступ в данном коде подразумевает, что оператор `else` предназначен для внешнего оператора `if`, т.е. он выполняется, когда значение `grade` заканчивается цифрой меньше 3. Однако, несмотря на наши намерения и вопреки отступу, ветвь `else` является частью внутреннего оператора `if`. Этот код добавляет '-' к оценкам, заканчивающимся на 3-7 включительно! Правильно выровненный, в соответствии с правилами выполнения, этот код выглядел бы так:

```
// отступ соответствует порядку выполнения,
// но не намерению программиста
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+'; // оценки, заканчивающиеся на
8 или 9,
                                // получают +
else
    lettergrade += '-'; // оценки, заканчивающиеся на
3, 4, 5, 6,
                                // получают - !
```

Контроль пути выполнения при помощи фигурных скобок

Заключив внутренний оператор `if` в блок, можно сделать ветвь `else` частью внешнего оператора `if`:

```
// добавлять плюс для оценок, заканчивающихся на 8
или 9, а минус для
// заканчивающихся на 0, 1 или 2
if (grade % 10 >= 3) {
    if (grade % 10 > 7)
        lettergrade += '+'; // оценки, заканчивающиеся на
8 или 9,
                                // получают +
} else
                                // скобки обеспечивают else
для внешнего if
    lettergrade += '-'; // оценки, заканчивающиеся на
0, 1 и 2,
                                // получают -
```

Операторы не распространяются за границы блока, поэтому внутренний цикл `if` заканчивается на закрывающей фигурной скобке

перед оператором `else`. Оператор `else` не может быть частью внутреннего оператора `if`. Теперь ближайшим свободным оператором `if` оказывается внешний, как и предполагалось изначально.

Упражнения раздела 5.3.1

Упражнение 5.5. Напишите собственную версию программы преобразования числовой оценки в символ с использованием оператора `if else`.

Упражнение 5.6. Перепишите программу оценки так, чтобы использовать условный оператор (см. раздел 4.7) вместо оператора `if else`.

Упражнение 5.7. Исправьте ошибки в каждом из следующих фрагментов кода:

- (a)

```
if (ival1 != ival2)
    ival1 = ival2
else ival1 = ival2 = 0;
```
- (b)

```
if (ival < minval)
    minval = 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();
```

Упражнение 5.8. Что такое "потерянный оператор `else`"? Как в языке C++ определяется принадлежность ветви `else`?

5.3.2. Оператор **switch**

Оператор switch предоставляет более удобный способ выбора одной из множества альтернатив. Предположим, например, что необходимо рассчитать, как часто встречается каждая из пяти гласных в некотором фрагменте текста. Программа будет иметь следующую логику.

- Читать каждый введенный символ.
- Сравнить каждый символ с набором искомых гласных.
- Если символ соответствует одной из гласных букв, добавить 1 к соответствующему счетчику.
- Отобразить результаты.

Программа должна отобразить результаты в следующем виде:

Number of vowel a: 3195

Number of vowel e: 6230

Number of vowel i: 3102

Number of vowel o: 3289

Number of vowel u: 1033

Для непосредственного решения этой задачи можно использовать оператор **switch**.

```
// инициализировать счетчики для каждой гласной
unsigned aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0,
uCnt = 0;
char ch;
while (cin >> ch) {
    // если ch - гласная, увеличить соответствующий
    счетчик
    switch (ch) {
        case 'a':
            ++aCnt;
            break;
        case 'e':
            ++eCnt;
            break;
        case 'i':
            ++iCnt;
            break;
        case 'o':
            ++oCnt;
            break;
        case 'u':
            ++uCnt;
            break;
    }
}
```

```

        break;
    case 'u':
        ++uCnt;
        break;
    }
}
// вывод результата
cout << "Number of vowel a: \t" << aCnt << '\n'
    << "Number of vowel e: \t" << eCnt << '\n'
    << "Number of vowel i: \t" << iCnt << '\n'
    << "Number of vowel o: \t" << oCnt << '\n'
    << "Number of vowel u: \t" << uCnt << endl;

```

Оператор `switch` вычисляет результат выражения, расположенного за ключевым словом `switch`. Это выражение может быть объявлением инициализированной переменной (см. раздел 5.2). Выражение преобразуется в целочисленный тип. Результат выражения сравнивается со значением, ассоциированным с каждым оператором `case`.

Если результат выражения соответствует значению метки `case`, выполнение кода начинается с первого оператора после *этой* метки. В принципе выполнение кода продолжается до конца оператора `switch`, но оно может быть прервано оператором `break`.

Более подробно оператор `break` рассматривается в разделе 5.5.1, а пока достаточно знать, что он прерывает текущий поток выполнения. В данном случае оператор `break` передает управление первому оператору после оператора `switch`. Здесь оператор `switch` является единственным оператором в теле цикла `while`, поэтому его прерывание возвращает контроль окружающему оператору `while`. Поскольку в нем нет никаких других операторов, цикл `while` продолжается, если его условие выполняется.

Если соответствия не найдено, выполнение сразу переходит к первому оператору после `switch`. Как уже упоминалось, в этом примере выход из оператора `switch` передает управление условию цикла `while`.

Ключевое слово `case` и связанное с ним значение называют также *меткой case* (*case label*). Значением каждой метки `case` является константное выражение (см. раздел 2.4.4).

```

char ch = getVal();
int ival = 42;
switch( ch) {

```

```
case 3.14: // ошибка: метка case не целое число
case ival: // ошибка: метка case не константа
// ...
```

Одинарные значения меток case недопустимы. Существует также специальная метка default, рассматриваемая ниже.

Порядок выполнения в операторе switch

Важно понимать, как управление передается между метками case. После обнаружения соответствующей метки case выполнение начинается с нее и продолжается далее через все остальные метки до конца или пока выполнение не будет прервано явно. Во избежание выполнения последующих разделов case выполнение следует прервать явно, поэтому оператор break обычно является последним оператором перед следующей меткой case.

Однако возможны ситуации, когда необходимо именно стандартное поведение оператора switch. У каждой метки case может быть только одно значение, однако две или более метки могут совместно использовать единый набор действий. В таких ситуациях достаточно пропустить оператор break и позволить программе пройти несколько меток case.

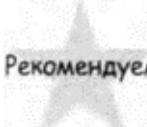
Например, можно было бы посчитать общее количество гласных так:

```
unsigned vowelCnt = 0;
// ...
switch (ch) {
    // для инкремента vowelCnt подойдет любая буква а,
    e, i, о или у
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        ++vowelCnt;
        break;
}
```

Здесь расположено несколько меток case подряд без оператора break. Теперь при любой гласной в переменной ch будет выполняться тот же код.

Поскольку язык C++ не требует обязательно располагать метки case в отдельной строке, весь диапазон значений можно указать в одной строке:

```
switch (ch) {  
    // альтернативный допустимый синтаксис  
    case 'a': case 'e': case 'i': case 'o': case 'u':  
        ++vowelCnt;  
        break;  
}
```



Рекомендуем

Случаи, когда оператор `break` пропускают преднамеренно, довольно редки, поэтому их следует обязательно комментировать, объясняя логику действий.

Пропуск оператора `break` — весьма распространенная ошибка

Весьма распространено заблуждение, что выполняются только те операторы, которые связаны с совпавшей меткой `case`. Вот пример неправильной реализации подсчета гласных в операторе `switch`:

```
// внимание: преднамеренно неправильный код!  
switch (ch) {  
    case 'a' :  
        ++aCnt; // Упс! Необходим оператор break  
    case 'e' :  
        ++eCnt; // Упс! Необходим оператор break  
    case 'i' :  
        ++iCnt; // Упс! Необходим оператор break  
    case 'o' :  
        ++oCnt; // Упс! Необходим оператор break  
    case 'u' :  
        ++uCnt;  
}
```

Чтобы понять происходящее, предположим, что значением переменной `ch` является '`e`'. Выполнение переходит к коду после метки `case 'e'`, где происходит инкремент переменной `eCnt`. Выполнение продолжается далее через метки `case`, увеличивая также значения переменных `iCnt`, `oCnt` и `uCnt`.



Рекомендуем

Несмотря на то что оператор `break` и не обязателен после последней метки оператора `switch`, использовать его все же рекомендуется. Ведь если впоследствии оператор `switch` будет дополнен еще одной меткой `case`, отсутствие оператора `break` после прежней последней метки не создаст проблем.

Метка default

Операторы после *метки default* выполняются, если ни одна из меток `case` не соответствует значению выражения оператора `switch`. Например, в рассматриваемый код можно добавить счетчик негласных букв. Значение этого счетчика по имени `otherCnt` будет увеличиваться в случае `default`:

```
// если ch гласная, увеличить соответствующий
счетчик
switch (ch) {
    case 'a': case 'e': case 'i': case 'o': case 'u':
        ++vowelCnt;
        break;
    default:
        ++otherCnt;
        break;
}
```

В этой версии, если переменная `ch` не содержит гласную букву, управление перейдет к метке `default` и увеличится значение счетчика `otherCnt`.



Рекомендуем

Раздел `default` имеет смысл создавать всегда, даже если в нем не происходит никаких действий. Впоследствии это однозначно укажет читателю кода, что случай `default` не был забыт, т.е. для остальных случаев никаких действий предпринимать не нужно.

Метка не может быть автономной; она должна предшествовать

оператору или другой метке case. Если оператор switch заканчивается разделом default, в котором не осуществляется никаких действий, за меткой default должен следовать пустой оператор или пустой блок.

Определение переменной в операторе switch

Как уже упоминалось, выполнение оператора switch способно переходить через метки case. Когда выполнение переходит к некой метке case, весь расположенный выше код оператора switch будет проигнорирован. Факт игнорирования кода поднимает интересный вопрос: что будет, если пропущенный код содержит определение переменной?

Ответ прост: недопустим переход с места, где переменная с инициализатором уже вышла из области видимости к месту, где эта переменная находится в области видимости.

```
case true:  
    // этот оператор switch недопустим, поскольку  
инициализацию  
    // можно обойти  
    string file_name; // ошибка: выполнение обходит  
неявно  
                                // инициализированную переменную  
    int ival = 0;          // ошибка: выполнение обходит  
неявно  
                                // инициализированную переменную  
    int jval;              // ok: поскольку jval не  
инициализирована  
    break;  
case false:  
    // ok: jval находится в области видимости, но она  
не инициализирована  
    jval = next_num();     // ok: присвоить значение  
jval  
    if (file_name.empty()) // file_name находится в  
области видимости, но  
                                // она не инициализирована  
    // ...
```

Если бы этот код был допустим, то любой переход к случаю false обходил бы инициализацию переменных file_name и ival, но они оставались бы в области видимости и код вполне мог бы использовать их.

Однако эти переменные не были бы инициализированы. В результате язык не позволяет перепрыгивать через инициализацию, если инициализированная переменная находится в области видимости в пункте, к которому переходит управление.

Если необходимо определить и инициализировать переменную для некоего случая case, то сделать это следует в блоке, гарантируя таким образом, что переменная выйдет из области видимости перед любой последующей меткой.

```
case true:  
{  
    // ok: оператор объявления в пределах  
    // операторного блока  
    string file_name = get_file_name();  
    // ...  
}  
break;  
case false:  
if (file_name.empty()) // ошибка: file_name вне  
// области видимости
```

Упражнения раздела 5.3.2

Упражнение 5.9. Напишите программу, использующую серию операторов if для подсчета количества гласных букв в тексте, прочитанном из потока cin.

Упражнение 5.10. Программа подсчета гласных имеет одну проблему: она не учитывает заглавные буквы как гласные. Напишите программу, которая подсчитывает гласные буквы как в верхнем, так и в нижнем регистре. То есть значение счетчика aCnt должно увеличиваться при встрече как символа 'a', так и символа 'A' (аналогично для остальных гласных букв).

Упражнение 5.11. Измените рассматриваемую программу так, чтобы она подсчитывала также количество пробелов, символов табуляции и новой строки.

Упражнение 5.12. Измените рассматриваемую программу так, чтобы она подсчитывала количество встреченных двухсимвольных последовательностей: ff, fl и fi.

Упражнение 5.13. Каждая из приведенных ниже программ содержит распространенную ошибку. Выявите и исправьте каждую из них.

Код для упражнения 5.13

```
( a) unsigned aCnt = 0, eCnt = 0, iouCnt = 0;
      char ch = next_text();
      switch (ch) {
        case 'a': aCnt++;
        case 'e': eCnt++;
        default: iouCnt++;
      }
( b) unsigned index = some_value();
      switch (index) {
        case 1:
          int ix = get_value();
          ivec[ix] = index;
          break;
        default:
          ix = ivec.size() - 1;
          ivec[ix] = index;
( c) unsigned evenCnt = 0, oddCnt = 0;
      int digit = get_num() % 10;
      switch (digit) {
        case 1, 3, 5, 7, 9:
          oddcnt++;
          break;
        case 2, 4, 6, 8, 10:
          evencnt++;
          break;
      }
( d) unsigned ival=512, jval=1024, kval=4096;
      unsigned bufsize;
      unsigned swt = get_bufCnt();
      switch(swt) {
        case ival:
          bufsize = ival * sizeof(int);
          break;
        case jval:
          bufsize = jval * sizeof(int);
          break;
        case kval:
          bufsize = kval * sizeof(int);
          break;
      }
```


5.4. Итерационные операторы

Итерационные операторы (iterative statement), называемые также *циклами* (loop), обеспечивают повторное выполнение кода, пока их условие истинно. Операторы `while` и `for` проверяют условие прежде, чем выполнить тело. Оператор `do while` сначала выполняет тело, а затем проверяет свое условие.



5.4.1. Оператор `while`

Оператор `while` многократно выполняет оператор, пока его условие остается истинным. Его синтаксическая форма имеет следующий вид:

`while (условие)`

оператор

Пока *условие* истинно (значение `true`), *оператор* (который зачастую является блоком кода) выполняется. Условие не может быть пустым. Если при первой проверке условие ложно (значение `false`), оператор не выполняется.

Условие может быть выражением или объявлением инициализированной переменной (см. раздел 5.2). Обычно либо само условие, либо тело цикла должно делать нечто изменяющее значение выражения. В противном случае цикл никогда не закончится.



Переменные, определенные в условии или теле оператора `while`, создаются и удаляются при каждой итерации.

Использование цикла `while`

Цикл `while` обычно используется в случае, когда итерации необходимо выполнять неопределенное количество раз, например, при чтении ввода. Цикл `while` полезен также при необходимости доступа к значению управляющей переменной после завершения цикла. Рассмотрим пример.

```

vector<int> v;
int i;
// читать до конца файла или отказа ввода
while (cin >> i)
    v.push_back(i); // найти первый отрицательный
Элемент
auto beg = v.begin();
while (beg != v.end() && *beg >= 0)
    ++beg;
if (beg == v.end())
    // известно, что все элементы v больше или равны
нулю

```

Первый цикл читает данные со стандартного устройства ввода. Он может выполняться сколько угодно раз. Условие становится ложным, когда поток `cin` читает недопустимые данные, происходит ошибка ввода или встречается конец файла. Второй цикл продолжается до тех пор, пока не будет найдено отрицательное значение. Когда цикл заканчивается, переменная `beg` будет либо равна `v.end()`, либо обозначит элемент вектора `v`, значение которого меньше нуля. Значение переменной `beg` можно использовать вне цикла `while` для дальнейшей обработки.

Упражнения раздела 5.4.1

Упражнение 5.14. Напишите программу для чтения строк со стандартного устройства ввода и поиска совпадающих слов. Программа должна находить во вводе места, где одно слово непосредственно сопровождается таким же. Отследите наибольшее количество повторений и повторяющееся слово. Отобразите максимальное количество дубликатов или сообщение, что никаких повторений не было. Например, при вводе `how now now now brown cow cow` вывод должен указать, что слово `now` встретилось три раза.



5.4.2. Традиционный оператор `for`

Оператор `for` имеет следующий синтаксис:

```

for ( инициализирующий-оператор условие; выражение )
    оператор

```

Слово `for` и часть в круглых скобках зачастую упоминают как заголовок `for` (for header).

Инициализирующий-оператор должен быть оператором объявления, выражением или пустым оператором. Каждый из этих операторов завершается точкой с запятой, поэтому данную синтаксическую форму можно рассматривать так:

```
for ( инициализатор; условие; выражение )
    оператор
```

Как правило, *инициализирующий-оператор* используется для инициализации или присвоения исходного значения переменной, изменяемой в цикле. Для управления циклом служит *условие*. Пока *условие* истинно, *оператор* выполняется. Если при первой проверке *условие* оказывается ложным, оператор не выполняется ни разу. Для изменения значения переменной, инициализированной в инициализирующем операторе и проверяемой в условии, используется *выражение*. Оно выполняется после каждой итерации цикла. Как и в других случаях, *оператор* может быть одиночным оператором или блоком операторов.

Поток выполнения в традиционном цикле `for`

Рассмотрим следующий цикл `for` из раздела 3.2.3:

```
// обрабатывать символы,
// пока они не исчерпаются или не встретится пробел
for ( decltype(s.size()) index = 0;
      index != s.size() && !isspace( s[ index ] );
++index)
    s[ index ] = toupper( s[ index ] ); // преобразовать в
верхний регистр
```

Порядок его выполнения таков.

1. В начале цикла только однажды выполняется *инициализирующий-оператор*. В данном случае определяется переменная `index` и инициализируется нулём.

2. Затем обрабатывается *условие*. Если `index` не равен `s.size()` и символ в элементе `s[index]` не является пробелом, то выполняется тело цикла `for`. В противном случае цикл заканчивается. Если условие ложно уже на первой итерации, то тело цикла `for` не выполняется вообще.

3. Если условие истинно, то тело цикла `for` выполняется. В данном случае оно переводит символ в элементе `s[index]` в верхний регистр.

4. И наконец, обрабатывается *выражение*. В данном случае значение переменной `index` увеличивается 1.

Эти четыре этапа представляют первую итерацию цикла `for`. Этап 1 выполняется только однажды при входе в цикл. Этапы 2–4 повторяются, пока условие не станет ложно, т.е. пока не встретится символ пробела в элементе `s` или пока `index` не превысит `s.size()`.



Не забывайте, что видимость любого объекта, определенного в пределах заголовка `for`, ограничивается телом цикла `for`. Таким образом, в данном примере переменная `index` недоступна после завершения цикла `for`.

Несколько определений в заголовке `for`

Подобно любому другому объявлению, *инициализирующий-оператор* способен определить несколько объектов. Однако только *инициализирующий-оператор* может быть оператором объявления. Поэтому у всех переменных должен быть тот же базовый тип (см. раздел 2.3). Для примера напишем цикл, дублирующий элементы вектора в конец следующим образом:

```
// запомнить размер v и остановиться,  
// достигнув первоначально последнего элемента  
for (decltype(v.size()) i = 0, sz = v.size(); i !=  
sz; ++i)  
    v.push_back(v[i]);
```

В этом цикле *инициализирующий-оператор* определяется индекс `i` и управляющая переменная цикла `sz`.

Пропуск частей заголовка `for`

В заголовке `for` может отсутствовать любой (или все) элемент: *инициализирующий-оператор*, *условие* или *выражение*.

Когда инициализация не нужна, вместо инициализирующего оператора можно использовать пустой оператор. Например, можно переписать цикл, который искал первое отрицательное число в векторе так, чтобы использовался цикл `for`:

```
auto beg = v.begin();  
for (/* ничего */; beg != v.end() && *beg >= 0;
```

```
++beg)  
; // ничего не делать
```

Обратите внимание: для указания на отсутствие инициализирующего оператора точка с запятой необходима, точнее, точка с запятой представляет пустой инициализирующий оператор. В этом цикле `for` тело также пусто, поскольку все его действия осуществляются в условии и выражении. Условие решает, когда придет время прекращать просмотр, а выражение увеличивает итератор.

Отсутствие части `условие` эквивалентно расположению в условии значения `true`. Поскольку условие истинно всегда, тело цикла `for` должно содержать оператор, обеспечивающий выход из цикла. В противном случае цикл будет выполняться бесконечно.

```
for (int i = 0; /* нет условия */ ; ++i) {  
    // обработка i; код в цикле должен остановить  
    итерацию!  
}
```

В заголовке `for` может также отсутствовать `выражение`. В таких циклах либо условие, либо тело должно делать нечто обеспечивающее итерацию. В качестве примера перепишем цикл `while`, читающий ввод в вектор целых чисел.

```
vector<int> v;  
for (int i; cin >> i; /* нет выражения */ )  
    v.push_back(i);
```

В этом цикле нет никакой необходимости в выражении, поскольку условие изменяет значение переменной `i`. Условие проверяет входной поток, поэтому цикл заканчивается, когда прочитан весь ввод или произошла ошибка ввода.

Упражнения раздела 5.4.2

Упражнение 5.15. Объясните каждый из следующих циклов. Исправьте все обнаруженные ошибки.

- (a) `for (int ix = 0; ix != sz; ++ix) { /* ... */ }`
`if (ix != sz)`
`// ...`
- (b) `int ix;`
`for (ix != sz; ++ix) { /* ... */ }`
- (c) `for (int ix = 0; ix != sz; ++ix, ++sz) { /* ... */ }`

Упражнение 5.16. Цикл `while` особенно хорош, когда необходимо выполнить некое условие; например, когда нужно читать значения до конца файла. Цикл `for` считают циклом пошагового выполнения: индекс проходит диапазон значений в коллекции. Напишите идиоматическое использование каждого цикла, а затем перепишите каждый случай использования в другой конструкции цикла. Если бы вы могли использовать только один цикл, то какой бы вы выбрали и почему?

Упражнение 5.17. Предположим, есть два вектора целых чисел. Напишите программу, определяющую, не является ли один вектор префиксом другого. Для векторов неравной длины сравнивайте количество элементов меньшего вектора. Например, если векторы содержат значения `0, 1, 1, 2` и `0, 1, 1, 2, 3, 5, 8` соответственно, ваша программа должна возвратить `true`.



5.4.3. Серийный оператор `for`



Новый стандарт ввел упрощенный оператор `for`, который перебирает элементы контейнера или другой последовательности. Синтаксис *серийного оператора for* (*range for*) таков:

```
for (объявление : выражение)
    оператор
```

выражение должно представить некую последовательность, такую, как список инициализации (см. раздел 3.3.1), массив (см. раздел 3.5), или объект такого типа, как `vector` или `string`, у которого есть функции-члены `begin()` и `end()`, возвращающие итераторы (см. раздел 3.4).

объявление определяет переменную. Каждый элемент последовательности должен допускать преобразование в тип переменной (см. раздел 4.11). Проще всего гарантировать соответствие типов за счет использования спецификатора типа `auto` (см. раздел 2.5.2). Так компилятор выведет тип сам. Если необходима запись в элементы последовательности, то переменная цикла должна иметь ссылочный тип.

На каждой итерации управляющая переменная определяется и инициализируется следующим значением последовательности, а затем

выполняется оператор. Как обычно, оператор может быть одиночным оператором или блоком. Выполнение завершается, когда все элементы обработаны.

Несколько таких циклов уже было представлено, но для завершенности рассмотрим цикл, удваивающий значение каждого элемента в векторе:

```
vector<int> v = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
// для записи в элементы переменная диапазона
должна быть ссылкой
for (auto &r : v) // для каждого элемента вектора v
    r *= 2;           // удвоить значение каждого
 элемента вектора v
```

Заголовок `for` объявляет, что управляющая переменная цикла `r` связана с вектором `v`. Чтобы позволить компилятору самостоятельно вывести тип переменной `r`, используем спецификатор `auto`. Поскольку предполагается изменение значений элементов вектора `v`, объявим переменную `r` как ссылку. При присвоении ей значений в цикле фактически присваивается значение элементу, с которым связана переменная `r` в данный момент.

Вот эквивалентное определение серийного оператора `for` в терминах традиционного цикла `for`:

```
for (auto beg = v.begin(), end = v.end(); beg != end; ++beg) {
    auto &r = *beg; // для изменения элементов r
должна быть ссылкой
    r *= 2;           // удвоить значение каждого
 элемента вектора v
}
```

Теперь, когда известно, как работает серийный оператор `for`, можно понять, почему в разделе 3.3.2 упоминалось о невозможности его использования для добавления элементов к вектору или другому контейнеру. В серийном операторе `for` кешируется значение `end()`. Если добавить или удалить элементы из последовательности, сохраненное значение `end()` станет неверным (см. раздел 3.4.1). Более подробная информация по этой теме приведена в разделе 9.3.6.

5.4.4. Оператор `do while`

Оператор `do while` похож на оператор `while`, но его условие

проверяется после выполнения тела. Независимо от значения условия тело цикла выполняется по крайней мере однажды. Его синтаксическая форма приведена ниже.

```
do  
    оператор  
    while ( условие );
```



После заключенного в скобки условия оператор `do while` заканчивается точкой с запятой.

В цикле `do while` оператор выполняется прежде, чем *условие*. Причем *условие* не может быть пустым. Если *условие* ложно, цикл завершается, в противном случае цикл повторяется. Используемые в условии переменные следует определить вне тела оператора `do while`.

Напишем программу, использующую цикл `do while` для суммирования любого количества чисел.

```
// многократно запрашивать у пользователя пары  
чисел для суммирования  
string rsp; // используется в условии, поэтому не  
может быть  
// определена в цикле do  
do {  
    cout << "please enter two values: ";  
    int val1 = 0, val2 = 0;  
    cin >> val1 >> val2;  
    cout << "The sum of " << val1 << " and " << val2  
        << " = " << val1 + val2 << "\n\n"  
        << "More? Enter yes or no: ";  
    cin >> rsp;  
} while (!rsp.empty() && rsp[0] != 'n');
```

Цикл начинается запросом у пользователя двух чисел. Затем выводится их сумма и следует запрос, желает ли пользователь суммировать далее. Ответ пользователя проверяется в условии. Если ввод пуст или начинается с `n`, цикл завершается. В противном случае цикл повторяется.

Поскольку условие не обрабатывается до окончания оператора или блока, цикл `do while` не позволяет определять переменные в условии.

```

do {
    // ...
    mumble( foo ) ;
} while ( int foo = get_foo( ) ); // ошибка:
объявление в условии do

```

Если определить переменные в условии, то любое их использование произойдет *прежде* определения!

Упражнения раздела 5.4.4

Упражнение 5.18. Объясните каждый из следующих циклов. Исправьте все обнаруженные ошибки.

```

(a) do
    int v1, v2;
    cout << "Please enter two numbers to sum: ";
    if ( cin >> v1 >> v2 )
        cout << "Sum is: " << v1 + v2 << endl;
    while ( cin );
(b) do {
    // ...
} while ( int ival = get_response( ) );
(c) do {
    int ival = get_response( );
} while ( ival );

```

Упражнение 5.19. Напишите программу, использующую цикл `do` `while` для циклического запроса у пользователя двух строк и указания, которая из них меньше другой.

5.5. Операторы перехода

Операторы перехода прерывают поток выполнения. Язык C++ предоставляет четыре оператора перехода: `break`, `continue` и `goto`, рассматриваемые в этой главе, и оператор `return`, который будет описан в разделе 6.3.

5.5.1. Оператор `break`

Оператор `break` завершает ближайший окружающий оператор `while`, `do while`, `for` или `switch`. Выполнение возобновляется с оператора, следующего непосредственно за завершающим оператором.

Оператор `break` может располагаться только в цикле или операторе `switch` (включая операторы или блоки, вложенные в эти циклы). Оператор `break` воздействует лишь на ближайший окружающий цикл или оператор `switch`.

```
string buf;
while (cin >> buf && !buf.empty( )) {
    switch( buf[ 0] ) {
        case '-':
            // продолжать до первого пробела
            for (auto it = buf.begin() + 1; it != buf.end(); ++it) {
                if (*it == ' ')
                    break; // #1, выйти из цикла for
                // ...
            }
            // break #1 передает управление сюда
            // дальнейшая обработка случая '-'
            break; // #2, выйти из оператора switch
        case '+':
            // ...
    } // конец оператора switch
    // break #2 передает управление сюда
} // конец оператора while
```

Оператор `break` с меткой #1 завершает цикл `for` в разделе `case` для случая дефиса. Он не завершает внешний оператор `switch` и даже не завершает обработку текущего случая. Выполнение продолжается с первого оператора после цикла `for`, который мог бы содержать дополнительный код обработки случая дефиса или оператор `break`, который завершает данный раздел.

Оператор `break` с меткой #2 завершает оператор `switch`, но не внешний цикл `while`. Выполнение кода после оператора `break` продолжает условие цикла `while`.

Упражнения раздела 5.5.1

Упражнение 5.20. Напишите программу, которая читает последовательность строк со стандартного устройства ввода до тех пор, пока не встретится повторяющееся слово или пока ввод слов не будет закончен. Для чтения текста по одному слову используйте цикл `while`.

Для выхода из цикла при встрече двух совпадающих слов подряд используйте оператор `break`. Выведите повторяющееся слово, если оно есть, а в противном случае отобразите сообщение, свидетельствующее о том, что повторяющихся слов нет.

5.5.2. Оператор `continue`

Оператор `continue` прерывает текущую итерацию ближайшего цикла и немедленно начинает следующую. Оператор `continue` может присутствовать только в циклах `for`, `while` или `do while`, включая операторы или блоки, вложенные в такие циклы. Подобно оператору `break`, оператор `continue` во вложенном цикле действует только на ближайший окружающий цикл. Однако, в отличие от оператора `break`, оператор `continue` может присутствовать в операторе `switch`, только если он встроен в итерационный оператор.

Оператор `continue` прерывает только текущую итерацию; выполнение остается в цикле. В случае цикла `while` или `do while` выполнение продолжается с оценки условия. В традиционном цикле `for` выполнение продолжается в выражении заголовка. В серийном операторе `for` выполнение продолжается с инициализации управляющей переменной следующим элементом последовательности.

Следующий цикл читает со стандартного устройства ввода по одному слову за раз. Обработаны будут только те слова, которые начинаются с символа подчеркивания. Для любого другого значения текущая итерация заканчивается.

```
string buf;
while (cin >> buf && !buf.empty()) {
    if (buf[0] != '_')
        continue; // получить другой ввод
    // все еще здесь? ввод начинается с '_', обработка
    buf...
}
```

Упражнения раздела 5.5.2

Упражнение 5.21. Переделайте программу из упражнения раздела 5.5.1 так, чтобы она искала дубликаты только тех слов, которые начинаются с прописной буквы.



5.5.3. Оператор `goto`

Оператор `goto` обеспечивает безусловный переход к другому оператору в той же функции.

Рекомендуем

Не нужно использовать операторы `goto`. Они затрудняют и понимание, и изменение программ.

Оператор `goto` имеет следующий синтаксис:

`goto метка;`

Метка (label) — это идентификатор, которым помечен оператор. *Помеченный оператор (labeled statement)* — это любой оператор, которому предшествует идентификатор, сопровождаемый двоеточием.

`end: return; // помеченный оператор;` может быть целью оператора `goto`

Метки независимы от имен, используемых для переменных и других идентификаторов. Следовательно, у метки может быть тот же идентификатор, что и у другой сущности в программе, не вступая в конфликт с другим одноименным идентификатором. Оператор `goto` и помеченный оператор, на который он передает управление, должны находиться в той же функции.

Подобно оператору `switch`, оператор `goto` не может передать управление из точки, где инициализированная переменная вышла из области видимости, в точку, где эта переменная находится в области видимости.

```
// ...
goto end;
int ix = 10; // ошибка: goto обходит определение
инициализированной
// переменной
end:
// ошибка: код здесь мог бы использовать ix,
// но goto обошел ее объявление
ix = 42;
```

Переход назад за уже выполненное определение вполне допустим. Переходя назад к точке перед определением переменная приведет к ее удалению и повторному созданию.

```
// переход назад через определение
// инициализированной переменной приемлем
begin:
int sz = get_size();
if (sz <= 0) {
    goto begin;
}
```

При выполнении оператора `goto` переменная `sz` удаляется, а затем она определяется и инициализируется снова, когда управление передается назад за ее определение после перехода к метке `begin`.

Упражнения раздела 5.5.3

Упражнение 5.22. Последний пример этого раздела, с переходом назад к метке `begin`, может быть написан лучше с использованием цикла. Перепишите код так, чтобы устраниТЬ оператор `goto`.



5.6. Блоки `try` и обработка исключений

Исключения (exception) — это аномалии времени выполнения, такие как потеря подключения к базе данных или ввод непредвиденных данных, которые нарушают нормальное функционирование программы^[3]. Реакция на аномальное поведение может быть одним из самых трудных этапов разработки любой системы.

Обработка исключений обычно используется в случае, когда некая часть программы обнаруживает проблему, с которой она не может справиться, причем проблема такова, что обнаружившая ее часть программы не может продолжить выполнение. В таких случаях обнаруживший проблему участок программы нуждается в способе сообщить о случившемся и о том, что он неспособен продолжить выполнение. Способ сообщения о проблеме не подразумевает знания о том, какая именно часть программы будет справляться с создавшейся ситуацией. Сообщив о случившемся, обнаружившая проблему часть кода прекращает работу.

Каждой части программы, способной передать исключение, соответствует другая часть, код которой способен обработать исключение, независимо от того, что произошло. Например, если проблема в недопустимом вводе, то часть обработки могла бы попросить пользователя ввести правильные данные. Если потеряна связь с базой данных, то часть обработки могла бы предупредить об этом пользователя.

Исключения обеспечивают взаимодействие частей программы, обнаруживающих проблему и решающих ее. Обработка исключений в языке C++ подразумевает следующее.

- *Оператор* `throw` используется частью кода обнаружившего проблему, с которой он не может справиться. Об операторе `throw` говорят, что он *передает* (raise) исключение.

- *Блок* `try` используется частью обработки исключения. Блок `try` начинается с *ключевого слова* `try` и завершается одной или несколькими *директивами* `catch` (catch clause). Исключения, переданные из кода, расположенного в блоке `try`, как правило, обрабатываются в одном из разделов `catch`. Поскольку разделы `catch` обрабатывают исключение, их называют также *обрабочиками исключений* (exception handler).

- Набор определенных в библиотеке *классов исключений* (exception class) используется для передачи информации о произошедшем между операторами `throw` и соответствующими разделами `catch`.

В остальной части этого раздела три компонента обработки исключений рассматриваются последовательно. Более подробная информация об исключениях приведена в разделе 18.1.

5.6.1. Оператор `throw`

Обнаруживающая часть программы использует оператор `throw` для передачи исключения. Он состоит из ключевого слова `throw`, сопровождаемого выражением. Выражение определяет тип передаваемого исключения. Оператор `throw`, как правило, завершается точкой с запятой, что делает его выражением.

Для примера вернемся к программе раздела 1.5.2, в которой суммируются два объекта класса `Sales_item`. Она проверяет, относятся ли обе прочитанные записи к одной книге. Если нет, она отображает сообщение об ошибке и завершает работу.

```
Sales_item item1, item2;
cin >> item1 >> item2;
// сначала проверить, представляют ли объекты item1
и item2
// одну и ту же книгу
if (item1.isbn() == item2.isbn()) {
    cout << item1 + item2 << endl;
    return 0; // свидетельство успеха
} else {
    cerr << "Data must refer to same ISBN"
        << endl;
    return -1; // свидетельство отказа
}
```

В более реалистичной программе суммирующая часть могла бы быть отделена от части, обеспечивающей взаимодействие с пользователем. В таком случае проверяющую часть можно было бы переписать так, чтобы она передавала исключение, а не возвращала свидетельство отказа.

```
// сначала проверить, представляют ли объекты item1
и item2
// одну и ту же книгу
```

```

if (item1.isbn() != item2.isbn())
    throw runtime_error("Data must refer to same
ISBN");
// если управление здесь, значит, ISBN совпадают
cout << item1 + item2 << endl;

```

Если теперь ISBN окажутся разными, будет передан объект исключения типа `runtime_error`. Передача исключения завершает работу текущей функции и передает управление обработчику, способному справиться с этой ошибкой.

Тип `runtime_error` является одним из типов исключения, определенных в заголовке `stdexcept` стандартной библиотеки. Более подробная информация по этой теме приведена в разделе 5.6.3. Объект класса `runtime_error` следует инициализировать объектом класса `string` или символьной строкой в стиле C (см. раздел 3.5.4). Эта строка представляет дополнительную информацию о проблеме.

5.6.2. Блок `try`

Блок `try` имеет следующий синтаксис:

```

try {
    операторы_программы
} catch ( объявление_исключения ) {
    операторы_обработчика
} catch ( объявление_исключения ) {
    операторы_обработчика
} // ...

```

Блок `try` начинается с ключевого слова `try`, за которым следует блок кода, заключенный в фигурные скобки.

Блок `try` сопровождается одним или несколькими блоками `catch`. Блок `catch` состоит из трех частей: ключевого слова `catch`, объявления (возможно, безымянного) объекта в круглых скобках (называется *объявлением исключения* (*exception declaration*)) и операторного блока. Когда объявление исключения в блоке `catch` совпадает с исключением, выполняется связанный с ним блок. По завершении выполнения кода обработчика управление переходит к оператору, следующему непосредственно после него.

Операторы_программы в блоке `try` являются обычными программными операторами, реализующими ее логику. Подобно любым

другим блокам кода, блоки `try` способны содержать любые операторы языка C++, включая объявления. Объявленные в блоке `try` переменные недоступны вне блока, в частности, они не доступны в блоках `catch`.

Создание обработчика

В приведенном выше примере, чтобы избежать суммирования двух объектов класса `Sales_item`, представляющих разные книги, использовался оператор `throw`. Предположим, что суммирующая объекты класса `Sales_item` часть программы отделена от части, взаимодействующей с пользователем. Эта часть могла бы содержать примерно такой код обработки исключения, переданного в блоке сложения.

```
while (cin >> item1 >> item2) {
    try {
        // код, который складывает два объекта класса
        Sales_item
        // если при сложении произойдет сбой, код
передаст
        // исключение runtime_error
    } catch (runtime_error err) {
        // напомнить пользователю, что ISBN слагаемых
объектов
        // должны совпадать
        cout << err.what()
            << "\nTry Again? Enter y or n" << endl;
        char c;
        cin >> c;
        if (!cin || c == 'n')
            break; // выход из цикла while
    }
}
```

В блоке `try` расположена обычная логика программы. Это сделано потому, что данная часть программы способна передать исключение типа `runtime_error`.

Данный блок `try` обладает одним разделом `catch`, который обрабатывает исключение типа `runtime_error`. Операторы в блоке после ключевого слова `catch` определяют действия, выполняемые в случае, если код в блоке `try` передаст исключение `runtime_error`. В

данном случае обработка подразумевает отображение сообщения об ошибке и запрос у пользователя разрешения на продолжение. Когда пользователь вводит символ 'n', цикл `while` завершается, в противном случае он продолжается и считывает два новых объекта класса `Sales_item`.

В сообщении об ошибке используется текст, возвращенный функцией `err.what()`. Поскольку известно, что классом объекта исключения `err` является `runtime_error`, нетрудно догадаться, что функция `what()` является членом (см. раздел 1.5.2) класса `runtime_error`. В каждом из библиотечных классов исключений определена функция-член `what()`, которая не получает никаких аргументов и возвращает символьную строку в стиле С (т.е. `const char*`). В случае класса `runtime_error` эта строка является копией строки, использованной при инициализации объекта класса `runtime_error`. Если описанный в предыдущем разделе код передаст исключение, то отображенное ранее `catch` сообщение об ошибке будет иметь следующий вид:

```
Data must refer to same ISBN  
Try Again? Enter y or n
```

При поиске обработчика выполнение функций прерывается

В сложных системах программа может пройти через несколько блоков `try` прежде, чем встретится с кодом, который передает исключение. Например, в блоке `try` может быть вызвана функция, в блоке `try` которой содержится вызов другой функции с ее собственным блоком `try`, и т.д.

Поиск обработчика осуществляется по цепочке обращений в обратном порядке. Сначала поиск обработчика исключения осуществляется в той функции, в которой оно было передано. Если соответствующего раздела `catch` не найдено, работа функции завершается, а поиск продолжается в той функции, которая вызвала функцию, в которой было передано исключение. Если и здесь соответствующий раздел `catch` не найден, эта функция также завершается, а поиск продолжается по цепочке вызовов дальше, пока обработчик исключения соответствующего типа не будет найден.

Если соответствующий раздел `catch` так и не будет найден, управление перейдет к библиотечной функции `terminate()`, которая определена в заголовке `exception`. Поведение этой функции зависит от системы, но обычно она завершает выполнение программы.

Исключения, которые были переданы в программах, не имеющих

блоков `try`, обрабатываются аналогично: в конце концов, без блоков `try` не может быть никаких обработчиков и ни для каких исключений, которые, однако, вполне могут быть переданы. В таком случае исключение приводит к вызову функции `terminate()`, которая (как правило) и завершает работу программы.

***Внимание! Написание устойчивого к исключениям кода —
довольно сложная задача***

Важно понимать, что исключения прерывают нормальный поток программы. В месте, где происходит исключение, некоторые из действий, ожидаемых вызывающей стороной, могут быть выполнены, а другие нет. Как правило, пропуск части программы может означать, что объект останется в недопустимом или неполном состоянии, либо что ресурс не будет освобожден и т.д. Программы, которые правильно "зачищают" объекты во время обработки исключений, называют *устойчивыми к исключениям* (exception safe). Написание устойчивого к исключениям кода чрезвычайно сложно и практически не рассматривается в данном вводном курсе.

Некоторые программы используют исключения просто для завершения программы в случае проблем. Такие программы вообще не заботятся об устойчивости к исключениям.

Программы, которые действительно обрабатывают исключения и продолжают работу, должны постоянно знать, какое исключение может произойти и что программа должна делать для гарантии допустимости объектов, невозможности утечки ресурсов и восстановления программы в корректном состоянии.

Некоторые из наиболее популярных методик обеспечения устойчивости к исключениям здесь будут упомянуты. Однако читатели, программы которых требуют надежной обработки исключений, должны знать, что рассматриваемых здесь методик недостаточно для полного обеспечения устойчивости к исключениям.

5.6.3. Стандартные исключения

В библиотеке C++ определен набор классов, объекты которых можно использовать для передачи сообщений о проблемах в функциях, определенных в стандартной библиотеке. Эти стандартные классы исключений могут быть также использованы в программах, создаваемых разработчиком. Библиотечные классы исключений определены в четырех

следующих заголовках.

- В заголовке `<exception>` определен общий класс исключения `exception`. Он сообщает только о том, что исключение произошло, но не предоставляет никакой дополнительной информации.

- В заголовке `<stdexcept>` определено несколько универсальных классов исключения (табл. 5.1).

- В заголовке `<new>` определен класс исключения `bad_alloc`, рассматриваемый в разделе 12.1.2.

- В заголовке `<type_info>` определен класс исключения `bad_cast`, рассматриваемый в разделе 19.2.

В классах `exception`, `bad_alloc` и `bad_cast` определен только стандартный конструктор (см. раздел 2.2.1), поэтому невозможно инициализировать объект этих типов.

Поведение исключений других типов прямо противоположно: их можно инициализировать объектом класса `string` или строкой в стиле C, однако значением по умолчанию их инициализировать нельзя. При создании объекта исключения любого из этих типов необходимо предоставить инициализатор. Этот инициализатор используется для предоставления дополнительной информации о произошедшей ошибке.

Таблица 5.1. Стандартные классы исключений, определенные в заголовке `<stdexcept>`

<code>exception</code>	Наиболее общий вид проблемы
<code>runtime_error</code>	Проблема, которая может быть обнаружена только во время выполнения
<code>range_error</code>	Ошибка времени выполнения: полученный результат превосходит допустимый диапазон значения
<code>overflow_error</code>	Ошибка времени выполнения: переполнение регистра при вычислении
<code>underflow_error</code>	Ошибка времени выполнения: недополнение регистра при вычислении
<code>logic_error</code>	Ошибка в логике программы
<code>domain_error</code>	Логическая ошибка: аргумент, для которого не существует результата
<code>invalid_argument</code>	Логическая ошибка: неподходящий аргумент
<code>length_error</code>	Логическая ошибка: попытка создать объект большего размера, чем максимально допустимый для данного типа
<code>out_of_range</code>	Логическая ошибка: используемое значение вне допустимого диапазона

В классах исключений определена только одна функция `what()`. Она не получает никаких аргументов и возвращает константный указатель на тип `char`. Это указатель на символьную строку в стиле С (см. раздел 3.5.4), содержащую текст описания переданного исключения.

Содержимое символьного массива (строки в стиле С), указатель на который возвращает функция `what()`, зависит от типа объекта исключения. Для типов, которым при инициализации передают строку класса `string`, функция `what()` возвращает строку. Что же касается других типов, то возвращаемое значение зависит от компилятора.

Упражнения раздела 5.6.3

Упражнение 5.23. Напишите программу, которая читает два целых числа со стандартного устройства ввода и выводит результат деления первого числа на второе.

Упражнение 5.24. Перепишите предыдущую программу так, чтобы она передавала исключение, если второе число — нуль. Проверьте свою программу с нулевым вводом, чтобы увидеть происходящее при отсутствии обработчика исключения.

Упражнение 5.25. Перепишите предыдущую программу так, чтобы использовать для обработки исключения блок `try`. Раздел `catch` должен отобразить сообщение и попросить пользователя ввести новое число и повторить код в блоке `try`.

Резюме

Язык C++ предоставляет довольно ограниченное количество операторов. Некоторые из них предназначены для управления потоком выполнения программы.

- Операторы `while`, `for` и `do while` позволяют реализовать итерационные циклы.
- Операторы `if` и `switch` позволяют реализовать условное выполнение.
 - Оператор `continue` останавливает текущую итерацию цикла.
 - Оператор `break` осуществляет принудительный выход из цикла или оператора `switch`.
 - Оператор `goto` передает управление помеченному оператору.
 - Операторы `try` и `catch` позволяют создать блок `try`, в который заключают операторы программы, потенциально способные передать исключение. Оператор `catch` начинает раздел обработчика исключения,

код которого предназначен для реакции на исключение определенного типа.

- Оператор `throw` позволяет передать исключение, обрабатываемое в соответствующем разделе `catch`.

- Оператор `return` останавливает выполнение функции. (Подробнее об этом — в главе 6.)

Кроме того, существуют операторы выражения и операторы объявления. Объявления и определения переменных были описаны в главе 2.

Термины

Блокtry. Блок, начинаемый ключевым словом `try` и содержащий один или несколько разделов `catch`. Если код в блоке `try` передаст исключение, а один из разделов `catch` соответствует типу этого исключения, то исключение будет обработано кодом данного обработчика. В противном случае исключение будет обработано во внешнем блоке `try`, но если и этого не произойдет, сработает функция `terminate()`, которая и завершит выполнение программы.

Блок (block). Последовательность любого количества операторов, заключенная в фигурные скобки. Блок операторов может быть использован везде, где ожидается один оператор.

Директива catch (catch clause). Состоит из ключевого слова `catch`, объявления исключения в круглых скобках и блока операторов. Код в разделе `catch` предназначен для обработки исключения, тип которого указан в объявлении.

Класс исключения (exception class). Набор определенных стандартной библиотекой классов, используемых для сообщения об ошибке. Универсальные классы исключений см. в табл. 5.1.

Метка case. Константное выражение (см. раздел 2.4.4), следующее за ключевым словом `case` в операторе `switch`. Метки `case` в том же операторе `switch` не могут иметь одинакового значения.

Метка default. Метка оператора `switch`, соответствующая любому значению условия, не указанному в метках `case` явно.

Обработчик исключения (exception handler). Код, реагирующий на исключение определенного типа, переданное из другой части программы. Синоним термина *директива catch*.

Обявление исключения (exception declaration). Обявление в разделе

`catch`. Определяет тип исключений, обрабатываемых данным обработчиком.

Оператор`break`. Завершает ближайший вложенный цикл или оператор `switch`. Передает управление первому оператору после завершенного цикла или оператора `switch`.

Оператор`continue`. Завершает текущую итерацию ближайшего вложенного цикла. Передает управление условию цикла `while`, оператору `do` или выражению в заголовке цикла `for`.

Оператор`do while`. Подобен оператору `while`, но условие проверяется в конце цикла, а не в начале. Тело цикла выполняется по крайней мере однажды.

Оператор`for`. Оператор цикла, обеспечивающий итерационное выполнение. Зачастую используется для повторения вычислений определенное количество раз.

Серийный оператор`for` (range `for`). Управляющий оператор, перебирающий значения указанной коллекции и выполняющий некую операцию с каждым из них.

Оператор`goto`. Оператор, осуществляющий безусловную передачу управления помеченному оператору в другом месте той же функции. Операторы `goto` нарушают последовательность выполнения операций программы, поэтому их следует избегать.

Оператор`if`. Условное выполнение кода на основании значения в условии. Если условие истинно (значение `true`), тело оператора `if` выполняется, в противном случае управление переходит к оператору, следующему после него.

Оператор`if...else`. Условное выполнение кода в разделе `if` или `else`, в зависимости от истинности значения условия.

Оператор`switch`. Оператор условного выполнения, который сначала вычисляет результат выражения, следующего за ключевым словом `switch`, а затем передает управление разделу `case`, метка которого совпадает с результатом выражения. Когда соответствующей метки нет, выполнение переходит к разделу `default` (если он есть) или к оператору, следующему за оператором `switch`, если раздела `default` нет.

Оператор`throw`. Оператор, прерывающий текущий поток выполнения. Каждый оператор `throw` передает объект, который переводит управление на ближайший раздел `catch`, способный обработать исключение данного класса.

Оператор`while`. Оператор цикла, который выполняет оператор тела

до тех пор, пока условие остается истинным (значение `true`). В зависимости от истинности значения условия оператор выполняется любое количество раз.

Оператор выражения (expression statement). Выражение завершается точкой с запятой. Оператор выражения обеспечивает выполнение действий в выражении.

Передача (raise, throwing). Выражение, которое прерывает текущий поток выполнения. Каждый оператор `throw` передает объект, переводящий управление на ближайший раздел `catch`, способный обработать исключение данного класса.

Помеченный оператор (labeled statement). Оператор, которому предшествует метка. *Метка* (label) — это идентификатор, сопровождаемый двоеточием. Метки используются независимо от других одноименных идентификаторов.

Потерянный оператор `else` (dangling else). Разговорный термин, используемый для описания проблемы, когда во вложенной конструкции операторов `if` больше, чем операторов `else`. В языке C++ оператор `else` всегда принадлежит ближайшему расположенному выше оператору `if`. Чтобы указать явно, какому из операторов `if` принадлежит конкретный оператор `else`, применяются фигурные скобки.

Пустой оператор (null statement). Пустой оператор представляет собой отдельный символ точки с запятой.

Составной оператор (compound statement). Синоним блока.

Управление потоком (flow of control). Управление последовательностью выполнения операций в программе.

Устойчивость к исключениям (exception safe). Термин, описывающий программы, которые ведут себя правильно при передаче исключения.

Функция `terminate()`. Библиотечная функция, вызываемая в случае, если исключение так и не было обработано. Обычно завершает выполнение программы.

Глава 6

Функции

В этой главе описано, как объявлять и определять функции. Здесь также обсуждается передача функции аргументов и возвращение из них полученных значений. В языке C++ функции могут быть перегружены, т.е. то же имя может быть использовано для нескольких разных функций. Мы рассмотрим и то, как перегрузить функции, и то, как компилятор выбирает из нескольких перегруженных функций ее соответствующую версию для конкретного вызова. Завершается глава описанием указателей на функции.

Функция (function) — это именованный блок кода. Запуск этого кода на выполнение осуществляется при вызове функции. Функция может получать любое количество аргументов и (обычно) возвращает результат. Функция может быть перегружена, следовательно, то же имя может относиться к нескольким разным функциям.



6.1. Основы функций

Определение функции (function definition) обычно состоит из типа возвращаемого значения (return type), имени, списка параметров (parameter) и тела функции. Параметры определяются в разделяемом запятыми списке, заключенном в круглые скобки. Выполняемые функцией действия определяются в блоке операторов (см. раздел 5.1), называемом *телом функции* (function body).

Для запуска кода функции используется *оператор вызова* (call operator), представляющий собой пару круглых скобок. Оператор вызова получает выражение, являющееся функцией или указателем на функцию. В круглых скобках располагается разделяемый запятыми список аргументов (argument). Аргументы используются для инициализации параметров функции. Тип вызываемого выражения — это тип возвращаемого значения функции.

Создание функции

В качестве примера напишем функцию вычисления факториала заданного числа. Факториал числа n является произведением чисел от 1 до n . Факториал 5, например, равен 120:

$$1 * 2 * 3 * 4 * 5 = 120$$

Эту функцию можно определить следующим образом:

```
// факториал val равен
// val * (val - 1) * (val - 2) ... * ((val - (val -
1)) * 1)
int fact(int val) {
    int ret = 1; // локальная переменная для
содержания результата по
                // мере его вычисления
    while (val > 1)
        ret *= val--;
    // присвоение ret произведения ret
* val
                // и декремент val
    return ret; // возвратить результат
}
```

Функции присвоено имя fact. Она получает один параметр типа int и возвращает значение типа int. В цикле while вычисляется факториал с использованием постфиксного оператора декремента (см. раздел 4.5), уменьшающего значение переменной val на 1 при каждой итерации. Оператор return выполняется в конце функции fact и возвращает значение переменной ret.

Вызов функции

Чтобы вызвать функцию fact(), следует предоставить ей значение типа int. Результатом вызова также будет значение типа int:

```
int main() {
    int j = fact(5); // j равно 120, т. е. результату
fact(5)
    cout << "5! is " << j << endl;
    return 0;
}
```

Вызов функции осуществляет два действия: он инициализирует параметры функции соответствующими аргументами и передает управление коду этой функции. При этом выполнение *вызывающей* (calling) функции приостанавливается и начинается выполнение *вызываемой* (called) функции.

Выполнение функции начинается с неявного определения и инициализации ее параметров. Таким образом, когда происходит вызов функции fact(), сначала создается переменная типа int по имени val. Эта переменная инициализируется аргументом, предоставленным при вызове функции fact(), которым в данном случае является 5.

Выполнение функции заканчивается оператором return. Как и вызов функции, оператор return осуществляет два действия: возвращает значение (если оно есть) и передает управление назад *вызывающей* функции. Возвращенное функцией значение используется для инициализации результата вызывающего выражения. Выполнение продолжается с остальной частью выражения, в составе которого осуществлялся вызов. Таким образом, вызов функции fact() эквивалентен следующему:

```
int val = 5; // инициализировать val из литерала 5
int ret = 1; // код из тела функции fact
while (val > 1)
    ret *= val--;
```

```
int j = ret; // инициализировать j копией ret
```

Параметры и аргументы

Аргументы — это инициализаторы для параметров функции. Первый аргумент инициализирует первый параметр, второй аргумент инициализирует второй параметр и т.д. Хотя порядок инициализации параметров аргументами известен, порядок обработки аргументов не гарантирован (см. раздел 4.1.3). Компилятор может вычислять аргументы в любом порядке по своему предпочтению.

Тип каждого аргумента должен совпадать с типом соответствующего параметра, как и тип любого инициализатора должен совпадать с типом объекта, который он инициализирует. Следует передать точно такое же количество аргументов, сколько у функции параметров. Поскольку каждый вызов гарантированно передаст столько аргументов, сколько у функции параметров, последние всегда будут инициализированы.

Поскольку у функции `fact()` один параметр типа `int`, при каждом ее вызове следует предоставить один аргумент, который может быть преобразован в тип `int` (см. раздел 4.11):

```
fact( "hello" );           // ошибка: неправильный тип
аргумента
fact();                   // ошибка: слишком мало аргументов
fact( 42,    10,    0 );   // ошибка: слишком много
аргументов
fact( 3.14 );            // ok: аргумент преобразуется в
int
```

Первый вызов терпит неудачу потому, что невозможно преобразование значения типа `const char*` в значение типа `int`. Второй и третий вызовы передают неправильные количества аргументов. Функцию `fact()` следует вызывать с одним аргументом; ее вызов с любым другим количеством аргументов будет ошибкой. Последний вызов допустим, поскольку значение типа `double` преобразуется в значение типа `int`. В этом случае аргумент неявно преобразуется в тип `int` (с усечением). После преобразования этот вызов эквивалентен следующему:

```
fact( 3 );
```

Список параметров функции

Список параметров функции может быть пустым, но он не может отсутствовать. При определении функции без параметров обычно используют пустой список параметров. Для совместимости с языком C

можно также использовать ключевое слово `void`, чтобы указать на отсутствие параметров:

```
void f1() { /* ... */ } // неявно указанный пустой
список параметров
void f2(void) { /* ... */ } // явно указанный
пустой список параметров
```

Список параметров, как правило, состоит из разделяемого запятыми списка параметров, каждый из которых выглядит как одиночное объявление. Даже когда типы двух параметров одинаковы, объявление следует повторить:

```
int f3(int v1, v2) { /* ... */ } // ошибка
int f4(int v1, int v2) { /* ... */ } // ok
```

Параметры не могут иметь одинаковые имена. Кроме того, локальные переменные даже в наиболее удаленной области видимости в функции не могут использовать имя, совпадающее с именем любого параметра.

Имена в определении функций не обязательны, но все параметры обычно именуют. Поэтому у каждого параметра обычно есть имя. Иногда у функций есть не используемые параметры. Такие параметры зачастую оставляют безымянными, указывая, что они не используются. Наличие безымянного параметра не изменяет количество аргументов, которые следует передать при вызове. Аргумент при вызове должен быть предоставлен для каждого параметра, даже если он не используется.

Тип возвращаемого значения функции

В качестве типа возвращаемого значения функции применимо большинство типов. В частности, типом возвращаемого значения может быть `void`, это означает, что функция не возвращает значения. Но типом возвращаемого значения не может быть массив (см. раздел 3.5) или функция. Однако функция может возвратить указатель на массив или функцию. Определение функции, возвращающей указатель (или ссылку) на массив, рассматривается в разделе 6.3.3, а указателя на функцию — в разделе 6.7.

Упражнения раздела 6.1

Упражнение 6.1. В чем разница между параметром и аргументом?

Упражнение 6.2. Укажите, какие из следующих функций ошибочны и почему. Предложите способ их исправления.

(a) `int f() {
 string s;`

```
// ...
    return s;
}
(b) f2( int i) { /* ... */ }
(c) int calc( int v1, int v1) /* ... */
(d) double square( double x) return x * x;
```

Упражнение 6.3. Напишите и проверьте собственную версию функции `fact()`.

Упражнение 6.4. Напишите взаимодействующую с пользователем функцию, которая запрашивает число и вычисляет его факториал. Вызовите эту функцию из функции `main()`.

Упражнение 6.5. Напишите функцию, возвращающую абсолютное значение ее аргумента.



6.1.1. Локальные объекты

В языке C++ имя имеет область видимости (см. раздел 2.2.4), а объекты — продолжительность существования (object lifetime). Обе эти концепции важно понимать.

- Область видимости имени — это *часть текста программы*, в которой имя видимо.
- Продолжительность существования объекта — это *время при выполнении программы*, когда объект существует.

Как уже упоминалось, тело функции — это блок операторов. Как обычно, блок формирует новую область видимости, в которой можно определять переменные. Параметры и переменные, определенные в теле функции, называются *локальными переменными* (local variable). Они являются локальными для данной функции и скрывают (hide) объявления того же имени во внешней области видимости.

Объекты, определенные вне любой из функций, существуют на протяжении выполнения программы. Такие объекты создаются при запуске программы и не удаляются до ее завершения. Продолжительность существования локальной переменной зависит от того, как она определена.

Автоматические объекты

Объекты, соответствующие обычным локальным переменным,

создаются при достижении процессом выполнения определения переменной в функции. Они удаляются, когда процесс выполнения достигает конца блока, в котором определена переменная. Объекты, существующие только во время выполнения блока, известны как *автоматические объекты* (*automatic object*). После выхода процесса выполнения из блока значения автоматических объектов, созданных в этом блоке, неопределены.

Параметры — это автоматические объекты. Место для параметров резервируется при запуске функции. Параметры определяются в пределах тела функции. Следовательно, они удаляются по завершении функции.

Автоматические объекты, соответствующие параметрам функции, инициализируются аргументами, переданными функции. Автоматические объекты, соответствующие локальным переменным, инициализируются, если их определение содержит инициализатор. В противном случае они инициализируются значением по умолчанию (см. раздел 2.2.1), а это значит, что значения неинициализированных локальных переменных встроенного типа неопределены.

Локальные статические объекты

Иногда полезно иметь локальную переменную, продолжительность существования которой не прерывается между вызовами функции. Чтобы получить такие объекты, при определении локальной переменной используют ключевое слово `static`. Каждый локальный статический объект (local static object) инициализируется прежде, чем выполнение достигнет определения объекта. Локальная статическая переменная не удаляется по завершении функции; она удаляется по завершении программы.

В качестве простого примера рассмотрим функцию, подсчитывающую количество своих вызовов:

```
size_t count_calls() {
    static size_t ctr = 0; // значение сохраняется
    между вызовами
    return ++ctr;
}
int main() {
    for (size_t i = 0; i != 10; ++i)
        cout << count_calls() << endl;
    return 0;
}
```

Эта программа выводит числа от 1 до 10 включительно.

Прежде чем процесс выполнения впервые достигнет определения переменной `ctr`, она уже будет создана и получит исходное значение 0. Каждый вызов осуществляет инкремент переменной `ctr` и возвращает ее новое значение. При каждом запуске функции `count_calls()` переменная `ctr` уже существует и имеет некое значение, возможно, оставленное последним вызовом функции. Поэтому при втором вызове значением переменной `ctr` будет 1, при третьем — 2 и т.д.

Если у локальной статической переменной нет явного инициализатора, она инициализируется значением по умолчанию (см. раздел 3.3.1), следовательно, локальные статические переменные встроенного типа инициализируются нулем.

Упражнения раздела 6.1.1

Упражнение 6.6. Объясните различия между параметром, локальной переменной и локальной статической переменной. Приведите пример функции, в которой каждая из них могла бы быть полезной.

Упражнение 6.7. Напишите функцию, которая возвращает значение 0 при первом вызове, а при каждом последующем вызове возвращает последовательно увеличивающиеся числа.



6.1.2. Объявление функций

Как и любое другое имя, имя функции должно быть объявлено прежде, чем его можно будет использовать. Подобно переменным (см. раздел 2.2.2), функция может быть определена только однажды, но объявлена может быть многократно. За одним исключением, которое будет описано в разделе 15.3, можно объявить функцию, которая не определяется до тех пор, пока она не будет использована.

Обявление функции подобно ее определению, но у объявления нет тела функции. В объявлении тело функции заменяет точка с запятой.

Поскольку у объявления функции нет тела, нет никакой необходимости в именах параметров. Поэтому имена параметров зачастую отсутствуют в объявлении. Хоть имена параметров и не обязательны, они зачастую используются, чтобы помочь пользователям функции понять ее назначение:

```
// имена параметров указывают, что операторы
обозначают диапазон
// выводимых значений
void print( vector<int>::const_iterator beg,
            vector<int>::const_iterator end);
```

Эти три элемента объявления (тип возвращаемого значения, имя функции и тип параметров) описывают *интерфейс* (interface) функции. Они задают всю информацию, необходимую для вызова функции. Объявление функции называют также *прототипом функции* (function prototype).

Объявления функций находятся в файлах заголовка

Напомним, что объявления переменных располагают в файлах заголовка (см. раздел 2.6.3), а определения — в файлах исходного кода. По тем же причинам функции должны быть объявлены в файлах заголовка и определены в файлах исходного кода.

Весьма соблазнительно (и вполне допустимо) размещать объявления функций непосредственно в каждом файле исходного кода, который использует функцию. Однако такой подход утомителен и приводит к ошибкам. Помещая объявления функций в файлы заголовка, можно гарантировать, что все объявления данной функции будут одинаковы. Если необходимо изменить интерфейс функции, достаточно модифицировать его только в одном объявлении.



Рекомендуем

Файл исходного кода, в котором функция *определенна*, должен подключать заголовок, в котором функция *объявлена*. Так компилятор сможет проверить соответствие определения и объявления.

Упражнения раздела 6.1.2

Упражнение 6.8. Напишите файл заголовка по имени Chapter6.h, содержащий объявления функций, написанных для упражнений раздела 6.1



6.1.3. Раздельная компиляция

По мере усложнения программ возникает необходимость хранить различные части программы в отдельных файлах. Например, функции, написанные для упражнений раздела 6.1, можно было бы сохранить в одном файле, а код, использующий их, в других файлах исходного кода. Язык C++ позволяет разделять программы на логические части, предоставляя средство, известное как *раздельная компиляция* (separate compilation). Раздельная компиляция позволяет разделять программы на несколько файлов, каждый из которых может быть откомпилирован независимо.

Компиляция и компоновка нескольких файлов исходного кода

Предположим, например, что определение функции `fact()` находится в файле `fact.cc`, а ее объявление — в файле заголовка `Chapter6.h`. Файл `fact.cc`, как и любой другой файл, использующий эту функцию, будет включать заголовок `Chapter6.h`. Функцию `main()`, вызывающую функцию `fact()`, будем хранить в еще одном файле `factMain.cc`.

Чтобы создать *исполнимый файл* (executable file), следует указать компилятору, где искать весь используемый код. Эти файлы можно было бы откомпилировать следующим образом:

```
$ CC factMain.cc fact.cc # generates factMain.exe  
or a.out  
$ CC factMain.cc fact.cc -o main # generates main  
or main.exe
```

где `CC` — имя компилятора; `$` — системная подсказка; `#` — начало комментария командной строки. Теперь можно запустить исполняемый файл, который выполнит нашу функцию `main()`.

Если бы изменен был только один из наших файлов исходного кода, то перекомпилировать достаточно было бы только тот файл, который был фактически изменен. Большинство компиляторов предоставляет возможность раздельной компиляции каждого файла. Обычно этот процесс создает файл с расширением `.obj` (на Windows) или `.o` (на UNIX), указывающим, что этот файл содержит *объектный код* (object code).

Компилятор позволяет *скомпоновать* (link) *объектные файлы* (object file) и получить исполняемый файл. На системе авторов раздельная компиляция программы осуществляется следующим образом:

```
$ CC -c factMain.cc # generates  
factMain.o  
$ CC -c fact.cc # generates fact.o
```

```
$ CC factMain.o fact.o # generates  
factMain.exe or a.out  
$ CC factMain.o fact.o -o main # generates main or  
main.exe
```

Сверьтесь с руководством пользователя вашего компилятора, чтобы уточнить, как именно компилировать и запускать программы, состоящие из нескольких файлов исходного кода.

Упражнения раздела 6.1.3

Упражнение 6.9. Напишите собственные версии файлов fact.cc и factMain.cc. Эти файлы должны включать заголовок Chapter6.h из упражнения предыдущего раздела. Используйте эти файлы чтобы понять, как ваш компилятор обеспечивает раздельную компиляцию.



6.2. Передача аргументов

Как уже упоминалось, при каждом вызове функции ее параметры создаются заново. Используемое для инициализации параметра значение предоставляет соответствующий аргумент, переданный при вызове.



Параметры инициализируются точно так же, как и обычные переменные.

Как и у любой другой переменной, взаимодействие параметра и его аргумента определяет тип параметра. Если параметр — ссылка (см. раздел 2.3.1), то параметр привязывается к своему аргументу. В противном случае, значение аргумента копируется.

Когда параметр — ссылка, говорят, что его аргумент *передается по ссылке* (pass by reference) или что функция *вызывается по ссылке* (call by reference). Подобно любой другой ссылке, ссылочный параметр — это только псевдоним объекта, к которому он привязан, т.е. ссылочный параметр — псевдоним своего аргумента.

Когда значение аргумента копируется, параметр и аргумент — независимые объекты. Говорят, что такие аргументы *передаются по значению* (pass by value) или что функция *вызывается по значению* (call by value).



6.2.1. Передача аргумента по значению

При инициализации переменной не ссылочного типа значение инициализатора копируется. Изменения значения переменной никак не влияют на инициализатор:

```
int n = 0; // обычная переменная типа int
int i = n; // i - копия значения переменной n
i = 42;    // значение i изменилось, значение n -
```

нет

Передача аргумента по значению осуществляется точно так же; что бы функция не сделала с параметром, на аргумент это не повлияет. Например, в функции `fact()` (см. раздел 6.1) происходит декремент параметра `val`:

```
ret *= val--; // декремент значения val
```

Хотя функция `fact()` изменила значение `val`, это изменение никак не повлияло на переданный ей аргумент. Вызов `fact(i)` не изменяет значение переменной `i`.

Параметры указателя

Указатели (см. раздел 2.3.2) ведут себя, как любой не ссылочный тип. При копировании указателя его значение копируется. После создания копии получается два отдельных указателя. Однако указатель обеспечивает косвенный доступ к объекту, на который он указывает. Значение этого объекта можно изменить при помощи указателя (см. раздел 2.3.2):

```
int n = 0, i = 42;
int *p = &n, *q = &i; // p указывает на n; q
указывает на i
*p = 42; // значение n изменилось,
значение p - нет
p = q; // теперь p указывает на i;
значения i и n
// неизменны
```

То же поведение характерно для указателей, являющихся параметрами:

```
// функция получает указатель и обнуляет значение,
на которое он
// указывает
void reset(int *ip) {
    *ip = 0; // изменяет значение объекта, на который
указывает ip
    ip = 0; // изменяет только локальную копию ip;
аргумент неизменен
}
```

После вызова функции `reset()` объект, на который указывает аргумент, будет обнулен, но сам аргумент-указатель не изменится:

```
int i = 42;
reset(&i); // изменяет значение
i, но не адрес
cout << "i = " << i << endl; // выводит i = 0
```

Программисты, привыкшие к языку С, зачастую используют параметры в виде указателей для доступа к объектам вне функции. В языке С++ для этого обычно используют ссылочные параметры.

Упражнения раздела 6.2.1

Упражнение 6.10. Напишите, используя указатели, функцию, меняющую значения двух целых чисел. Проверьте функцию, вызвав ее и отобразив измененные значения.



6.2.2. Передача аргумента по ссылке

Напомним, что операции со ссылками — это фактически операции с объектами, к которым они привязаны (см. раздел 2.3.1):

```
int n = 0, i = 42;
int &r = n; // r привязан к n (т. е. r - другое имя
для n)
r = 42;      // теперь n = 42
r = i;       // теперь n имеет то же значение, что и
i
i = r;       // i имеет то же значение, что и n
```

Ссылочные параметры используют это поведение. Обычно они применяются, чтобы позволить функции изменить значение одного или нескольких аргументов.

Для примера можно переписать программу `reset` из предыдущего раздела так, чтобы использовать ссылку вместо указателя:

```
// функция, получающая ссылку на объект типа int и
обнуляющая его
void reset(int &i) // i - только другое имя
объекта, переданного
                    // на обнуление
{
    i = 0; // изменяет значение объекта, на который
    ссылается i
```

```
}
```

Подобно любой другой ссылке, ссылочный параметр связывается непосредственно с объектом, которым он инициализируется. При вызове этой версии функции `reset()` параметр `i` будет связан с любым переданным ей объектом типа `int`. Как и с любой ссылкой, изменения, сделанные с параметром `i`, осуществляются с объектом, на который она ссылается. В данном случае этот объект — аргумент функции `reset()`.

Когда вызывается эта версия функции `reset()`, объект передается непосредственно; поэтому нет никакой необходимости в передаче его адреса:

```
int j = 42;
reset(j); // j передается по ссылке; значение в j
изменяется
cout << "j = " << j << endl; // выводит j = 0
```

В этом вызове параметр `i` — это только другое имя переменной `j`. Любое использование параметра `i` в функции `reset()` фактически является использованием переменной `j`.

Использование ссылки во избежание копирования

Копирование объектов больших классов или больших контейнеров снижает эффективность программы. Кроме того, некоторые классы (включая классы I/O) не допускают копирования. Для работы с объектами, тип которых не допускает копирования, функции должны использовать ссылочные параметры.

В качестве примера напишем функцию сравнения длин двух строк. Поскольку строки могут быть очень длинными и их копирования желательно избежать, сделаем параметры ссылками. Так как сравнение двух строк не подразумевает их изменения, сделаем ссылочные параметры константами (см. раздел 2.4.1):

```
// сравнить длины двух строк
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

Как будет продемонстрировано в разделе 6.2.3, для не подлежащих изменению ссылочных параметров функции должны использовать ссылки на константу.



Ссылочные параметры, которые не изменяются в функции, должны быть объявлены как `const`.

Использование ссылочных параметров для возвращения дополнительной информации

Функция может возвратить только одно значение. Но что если функции нужно возвратить больше одного значения? Ссылочные параметры позволяют возвратить несколько результатов. В качестве примера определим функцию `find_char()`, которая возвращает позицию первого вхождения заданного символа в строке. Функция должна также возвращать количество этих символов в строке.

Как же определить функцию, возвращающую и позицию, и количество вхождений? Можно было бы определить новый тип, содержащий позицию и количество. Однако куда проще передать дополнительный ссылочный аргумент, содержащий количество вхождений:

```
// возвращает индекс первого вхождения с в s
// ссылочный параметр occurs содержит количество
// вхождений
string::size_type find_char(const string &s, char
c,
                           string::size_type
&occurs) {
    auto ret = s.size(); // позиция первого вхождения,
если оно есть
    occurs = 0;           // установить параметр
количество вхождений
    for (decltype(ret) i = 0; i != s.size(); ++i) {
        if (s[i] == c) {
            if (ret == s.size())
                ret = i; // запомнить первое вхождение с
            ++occurs; // инкремент счетчика вхождений
        }
    }
    return ret; // количество возвращается неявно в
параметре occurs
```

```
}
```

Когда происходит вызов функции `find_char()`, ей передаются три аргумента: строка, в которой осуществляется поиск, искомый символ и объект типа `size_type` (раздел 3.2.2), содержащий счетчик вхождений. Если `s` является объектом класса `string`, а `ctr` — объектом типа `size_type`, то функцию `find_char()` можно вызвать следующим образом:

```
auto index = find_char(s, 'o', ctr);
```

После вызова значением объекта `ctr` будет количество вхождений символа `o`, а `index` укажет на его первое вхождение, если оно будет. В противном случае значение `index` будет равно `s.size()`, а `ctr` — нулю.

Упражнения раздела 6.2.2

Упражнение 6.11. Напишите и проверьте собственную версию функции `reset()`, получающую ссылку.

Упражнение 6.12. Перепишите программу из упражнения 6.10 раздела 6.2.1 так, чтобы использовать ссылки вместо указателей при смене значений двух целочисленных переменных. Какая из версий, по вашему, проще в использовании и почему?

Упражнение 6.13. Если `T` — имя типа, объясните различие между функцией, объявленной как `void f(T)` и как `void f(T&)`.

Упражнение 6.14. Приведите пример, когда параметр должен быть ссылочным типом. Приведите пример случая, когда параметр не должен быть ссылкой.

Упражнение 6.15. Объясните смысл каждого из типов параметров функции `find_char()`. В частности, почему `s` — ссылка на константу, а `occurs` — простая ссылка? Почему эти параметры ссылочные, а параметр `c` типа `char` нет? Что будет, сделай мы `s` простой ссылкой? Что если `occurs` сделать константной ссылкой?

6.2.3. Константные параметры и аргументы

При использовании параметров, являющихся константой, следует помнить об обсуждении спецификатора `const` верхнего уровня из раздела 2.4.3. Как упоминалось в этом разделе, спецификатор `const` верхнего уровня — это тот спецификатор, который относится непосредственно к объекту:

```
const int ci = 42; // нельзя изменить ci; const
```

```
верхнего уровня
    int i = ci;                      // ok: при копировании ci
спецификатор const
                                         // верхнего уровня игнорируется
    int * const p = &i; // const верхнего уровня;
нельзя присвоить p
    *p = 0;                         // ok: изменение при помощи p
возможно; i теперь 0
```

Как и при любой другой инициализации, при копировании аргумента для инициализации параметра спецификаторы `const` верхнего уровня игнорируются. В результате спецификатор `const` верхнего уровня для параметров игнорируется. Параметру, у которого есть спецификатор `const` верхнего уровня, можно передать и константный, и неконстантный объект:

```
void fcn(const int i) { /* fcn может читать, но не
писать в i */ }
```

Функцию `fcn()` можно вызвать, передав ей аргумент типа `const int` или обычного типа `int`. Тот факт, что спецификаторы `const` верхнего уровня игнорируются у параметра, может иметь удивительные последствия:

```
void fcn(const int i) { /* fcn может читать, но не
писать в i */ }
void fcn(int i) { /* ... */ } // ошибка:
переопределяет fcn(int)
```

В языке C++ можно определить несколько разных функций с одинаковым именем. Однако это возможно только при достаточно большом различии их списков параметров. Поскольку спецификаторы `const` верхнего уровня игнорируются, мы можем передать те же типы любой версии функции `fcn()`. Вторая версия функции `fcn()` является ошибкой. Несмотря на внешний вид, ее список параметров не отличается от списка первой версии функции `fcn()`.

Параметры в виде указателей или ссылок и константность

Поскольку параметры инициализируются так же, как и переменные, имеет смысл напомнить общие правила инициализации. Можно инициализировать объект со спецификатором `const` нижнего уровня неконстантным объектом, но не наоборот, а простую ссылку следует инициализировать объектом того же типа.

```

int i = 42;
const int *cp = &i; // ok: но cp не может изменить
i (раздел 2.4.2)
const int &r = i; // ok: но r не может изменить i
(раздел 2.4.1)
const int &r2 = 42; // ok: (раздел 2.4.1)
int *p = cp; // ошибка: типы p и cp не совпадают
(раздел 2.4.2)
int &r3 = r; // ошибка: типы r3 и r не совпадают
(раздел 2.4.1)
int &r4 = 42; // ошибка: нельзя инициализировать
простую ссылку из
// литерала (раздел 2.3.1)

```

Те же правила инициализации относятся и к передаче параметров:

```

int i = 0;
const int ci = i;
string::size_type ctr = 0;
reset(&i); // вызывает версию функции reset с
параметром типа int*
reset(&ci); // ошибка: нельзя инициализировать int*
из указателя на
// объект const int
reset(i); // вызывает версию функции reset с
параметром типа int&
reset(ci); // ошибка: нельзя привязать простую
ссылку к константному
// объекту ci
reset(42); // ошибка: нельзя привязать простую
ссылку к литералу
reset(ctr); // ошибка: типы не совпадают; ctr имеет
беззнаковый тип
// ok: первый параметр find_char является ссылкой
на константу
find_char("Hello World!", 'o', ctr);

```

Ссылочную версию функции `reset()` (см. раздел 6.2.2) можно вызвать только для объектов типа `int`. Нельзя передать литерал, выражение, результат которого будет иметь тип `int`, объект, который требует преобразования, или объект типа `const int`. Точно так же

версии функции `reset()` с указателем можно передать только объект типа `int*` (см. раздел 6.2.1). С другой стороны, можно передать строковый литерал как первый аргумент функции `find_char()` (см. раздел 6.2.2). Ссылочный параметр этой функции — ссылка на константу, и можно инициализировать ссылки на константу из литералов.



По возможности используйте ссылки на константы

Весьма распространена ошибка, когда не изменяемые функцией параметры определяют как простые ссылки. Это создает у вызывающей стороны функции ложное впечатление, что функция могла бы изменить значение своего аргумента. Кроме того, использование ссылки вместо ссылки на константу неоправданно ограничивает типы аргументов, применяемые функцией. Как уже упоминалось, нельзя передать константный объект, литерал или требующий преобразования объект как простой ссылочный параметр.

В качестве примера рассмотрим функцию `find_char()` из раздела 6.2.2. Строковый параметр этой функции правильно сделан ссылкой на константу. Если бы этот параметр был определен как `string&`:

```
// ошибка: первый параметр должен быть const
string&
string::size_type find_char(string &s, char c,
                           string::size_type
&occurs);
```

то вызвать ее можно было бы только для объекта класса `string`, так что

```
find_char("Hello World", 'o', ctr);
```

привело бы к неудаче во времени компиляции.

Более того, эту версию функции `find_char()` нельзя использовать из других функций, которые правильно определяют свои параметры как ссылки на константу. Например, мы могли бы использовать функцию `find_char()` в функции, которая определяет, является ли строка предложением:

```
bool is_sentence(const string &s) {
    // если в конце s есть точка, то строка s -
    предложение
    string::size_type ctr = 0;
```

```
        return find_char(s, ctr) == s.size() - 1 && ctr ==  
1;  
    }
```

Если бы функция `find_char()` получала простую ссылку `string?`, то этот ее вызов привел бы к ошибке при компиляции. Проблема в том, что `s` — ссылка на `const string`, но функция `find_char()` была неправильно определена как получающая простую ссылку.

Было бы заманчиво попытаться исправить эту проблему, изменив тип параметра в функции `is_sentence()`. Но это только распространит ошибку, так как вызывающая сторона функции `is_sentence()` сможет передавать только неконстантные строки.

Правильный способ решения этой проблемы — исправить параметр функции `find_char()`. Если невозможно изменить функцию `find_char()`, определите локальную копию строки `s` в функции `is_sentence()` и передавайте эту строку функции `find_char()`.

Упражнения раздела 6.2.3

Упражнение 6.16. Несмотря на то что следующая функция допустима, она менее полезна, чем могла бы быть. Выявите и исправьте ограничение этой функции:

```
bool is_empty(string& s) { return s.empty(); }
```

Упражнение 6.17. Напишите функцию, определяющую, содержит ли строка какие-нибудь заглавные буквы. Напишите функцию, переводящую всю строку в нижний регистр. Использованные в этих функциях параметры имеют тот же тип? Если да, то почему? Если нет, то тоже почему?

Упражнение 6.18. Напишите объявления для каждой из следующих функций. Написав объявления, используйте имя функции для обозначения того, что она делает.

(а) Функция `compare()` возвращает значение типа `bool` и получает два параметра, являющиеся ссылками на класс `matrix`.

(б) Функция `change_val()` возвращает итератор `vector<int>` и получает два параметра: один типа `int`, а второй итератор для вектора `vector<int>`.

Упражнение 6.19. С учетом следующего объявления определите, какие вызовы допустимы, а какие нет. Объясните, почему они недопустимы.

```
double calc(double);  
int count(const string &, char);  
int sum(vector<int>::iterator,
```

```
vector<int>::iterator, int);  
vector<int> vec(10);  
( a) calc(23.4, 55.1); ( b) count("abcd", 'a');  
( c) calc(66); ( d) sum(vec.begin(),  
vec.end(), 3.8);
```

Упражнение 6.20. Когда ссылочные параметры должны быть ссылками на константу? Что будет, если сделать параметр простой ссылкой, когда это могла быть ссылка на константу?

6.2.4. Параметры в виде массива

Массивы обладают двумя особенностями, влияющими на определение и использование функций, работающих с массивами: массив нельзя скопировать (см. раздел 3.5.1), имя массива при использовании автоматически преобразуется в указатель на его первый элемент (см. раздел 3.5.3). Поскольку копировать массив нельзя, его нельзя передать функции по значению. Так как имя массива автоматически преобразуется в указатель, при передаче массива функции фактически передается указатель на его первый элемент.

Хотя передать массив по значению нельзя, вполне можно написать параметр, который выглядит как массив:

```
// несмотря на внешний вид,  
// эти три объявления функции print эквивалентны  
// у каждой функции есть один параметр типа const  
int*  
void print(const int*);  
void print(const int[]); // демонстрация  
намерения получить массив  
void print(const int[10]); // размерность только  
для документирования
```

Независимо от внешнего вида, эти объявления эквивалентны: в каждом объявлена функция с одним параметром типа `const int*`. Когда компилятор проверяет вызов функции `print()`, он выясняет только то, что типом аргумента является `const int*`:

```
int i = 0, j[2] = {0, 1};  
print(&i); // ok: &i - int*  
print(j); // ok: j преобразуется в int*,  
указывающий на j[0]
```

Если передать массив функции `print()`, то этот аргумент автоматически преобразуется в указатель на первый элемент в массиве; размер массива не имеет значения.



Подобно любому коду, который использует массивы, функции, получающие в качестве параметров массив, должны гарантировать

невыход за пределы его границ.

Поскольку массивы передаются как указатели, их размер функции обычно неизвестен. Они должны полагаться на дополнительную информацию, предоставляемую вызывающей стороной. Для управления параметрами указателя обычно используются три подхода.

Использование маркера для определения продолжения массива

Первый подход к управлению аргументами в виде массива требует, чтобы массив сам содержал маркер конца. Примером этого подхода являются символьные строки в стиле С (см. раздел 3.5.4). Строки в стиле С хранятся в символьных массивах, последний символ которых является нулевым. Функции, работающие со строками в стиле С, прекращают обработку массива, когда встречают нулевой символ:

```
void print( const char *cp) {  
    if ( cp)      // если cp не нулевой указатель  
        while ( *cp) // пока указываемый символ не  
        является нулевым  
            cout << *cp++; // вывести символ и перевести  
указатель  
    }  
}
```

Это соглашение хорошо работает с данными, где есть очевидное значение конечного маркера (такое, как нулевой символ), который не встречается в обычных данных. Это работает значительно хуже с такими данными, как целые числа, где каждое значение в диапазоне вполне допустимо.

Использование соглашения стандартной библиотеки

Второй подход обычно используется для управления аргументами в виде массива при передаче указателей на первый и следующий после последнего элемент массива. Подобный подход используется в стандартной библиотеке. Подробно этот стиль программирования обсуждается в части II. Используя этот подход, элементы массива можно отобразить следующим образом:

```
void print( const int *beg, const int *end) {  
    // вывести все элементы, начиная с beg и до, но не  
включая, end  
    while ( beg != end)  
        cout << *beg++ << endl; // вывести текущий
```

Элемент

```
// и перевести указатель  
}
```

Для вывода текущего элемента и перевода указателя `beg` на следующий элемент массива цикл `while` использует операторы обращения к значению и постфиксного инкремента (см. раздел 4.5). Цикл останавливается, когда `beg` становится равен `end`.

При вызове этой функции передаются два указателя: один на первый подлежащий отображению элемент и один на элемент после последнего:

```
int j[ 2 ] = { 0, 1 };  
// j преобразуется в указатель на первый элемент  
// массива j  
// второй аргумент - указатель на следующий элемент  
// после конца j  
print( begin( j ), end( j ) ); // функции begin и end см.  
р. 3.5.3
```

Эта функция безопасна, показывающая сторона правильно вычисляет указатели. Здесь эти указатели предоставляют библиотечные функции `begin()` и `end()` (см. раздел 3.5.3).

Явная передача параметра размера

Третий подход распространен в программах С и устаревших программах C++. Он подразумевает определение второго параметра, указывающего размер массива. Используя этот подход, перепишем функцию `print()` следующим образом:

```
// const int ia[] - эквивалент const int* ia  
// размер передается явно и используется для  
контроля доступа  
// к элементам ia  
void print( const int ia[], size_t size) {  
    for (size_t i = 0; i != size; ++i) {  
        cout << ia[ i ] << endl;  
    }  
}
```

Эта версия использует параметр `size` для определения количества выводимых элементов. Когда происходит вызов функции `print()`, ей следует передать этот дополнительный параметр:

```
int j[ ] = { 0, 1 }; // массив типа int размером 2
```

```
print( j, end( j ) - begin( j ) );
```

Функция безопасна, пока переданный размер не превосходит реальную величину массива.

Параметры массива и константность

Обратите внимание, что все три версии функции `print()` определяли свои параметры массива как указатели на константу. В разделе 6.2.3 было упомянуто о схожести указателей и ссылок. Когда функция не нуждается в записи элементов массива, параметр массива должен быть указателем на константу (см. раздел 2.4.2). Параметр должен быть простым указателем на неконстантный тип, только если функция должна изменять значения элементов.

Сылочный параметр массива

Подобно тому, как можно определить переменную, являющуюся ссылкой на массив (см. раздел 3.5.1), можно определить параметр, являющийся ссылкой на массив. Как обычно, сырочный параметр привязан к соответствующему аргументу, которым в данном случае является массив:

```
// ok: параметр является ссылкой на массив;
размерность - часть типа
void print( int ( &arr )[ 10 ] ) {
    for ( auto elem : arr )
        cout << elem << endl;
}
```



Круглые скобки вокруг части `&arr` необходимы (см. раздел 3.5.1):

```
f( int &arr[ 10 ] )      // ошибка: объявляет arr как
 массив ссылок
f( int ( &arr )[ 10 ] ) // ok: arr - ссылка на массив из
десети целых чисел
```

Поскольку размер массива является частью его типа, на размерность в теле функции вполне можно положиться. Однако тот факт, что размер является частью типа, ограничивает полноценность этой версии функции `print()`. Этую функцию можно вызвать только для массива из десяти

целых чисел:

```
int i = 0, j[ 2 ] = { 0, 1 };
int k[ 10 ] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
print( &i ); // ошибка: аргумент не массив из десяти
целых чисел
print( j ); // ошибка: аргумент не массив из десяти
целых чисел
print( k ); // ok: аргумент массив из десяти целых
чисел
```

В разделе 16.1.1 будет показано, как можно написать эту функцию способом, позволяющим передавать ссылочный параметр массива любого размера.

Передача многомерного массива

Напомним, что в языке C++ нет многомерных массивов (см. раздел 3.6). Вместо многомерных массивов есть массив массивов.

Подобно любому массиву, многомерный массив передается как указатель на его первый элемент (см. раздел 3.6). Поскольку речь идет о массиве массивов, элемент которого сам является массивом, указатель является указателем на массив. Размер второй размерности (и любой последующий) является частью типа элемента и должен быть определен:

```
// matrix указывает на первый элемент массива,
элементы которого
// являются массивами из десяти целых чисел
void print( int ( *matrix )[ 10 ], int rowSize ) { /* ... */
/* }
```

Объявляет `matrix` указателем на массив из десяти целых чисел.



Круглые скобки вокруг `*matrix` снова необходимы:

```
int *matrix[ 10 ]; // массив из десяти указателей
int ( *matrix )[ 10 ]; // указатель на массив из десяти
целых чисел
```

Функцию можно также определить с использованием синтаксиса массива. Как обычно, компилятор игнорирует первую размерность, таким образом, лучше не включать ее:

```
// эквивалентное определение
void print (int matrix[ ][10], int rowSize) { /* ...
*/ }
```

Здесь объявление `matrix` выглядит как двумерный массив. Фактически параметр является указателем на массив из десяти целых чисел.

Упражнения раздела 6.2.4

Упражнение 6.21. Напишите функцию, получающую значение типа `int` и указатель на тип `int`, а возвращающую значение типа `int`, если оно больше, или значение, на которое указывает указатель, если больше оно. Какой тип следует использовать для указателя?

Упражнение 6.22. Напишите функцию, меняющую местами два указателя на тип `int`.

Упражнение 6.23. Напишите собственные версии каждой из функций `print()`, представленных в этом разделе. Вызовите каждую из этих функций для вывода `i` и `j`, определенных следующим образом:

```
int i = 0, j[2] = {0, 1};
```

Упражнение 6.24. Объясните поведение следующей функции. Если в коде есть проблемы, объясните, где они и как их исправить.

```
void print( const int ia[ 10 ] ) {
    for ( size_t i = 0; i != 10; ++i )
        cout << ia[ i ] << endl;
}
```

6.2.5. Функция `main()`: обработка параметров командной строки

Функция `main()` — хороший пример того, как программы на языке C++ передают массивы в функции. До сих пор функция `main()` в примерах определялась с пустым списком параметров.

```
int main() { ... }
```

Но зачастую функции `main()` необходимо передать аргументы. Обычно аргументы функции `main()` используются для того, чтобы позволить пользователю задать набор параметров, влияющих на работу программы. Предположим, например, что функция `main()` программы находится в исполняемом файле по имени `prog`. Параметры программе можно передавать следующим образом:

```
prog -d -o ofile data0
```

Так, параметры командной строки передаются функции `main()` в двух (необязательных) параметрах:

```
int main( int argc, char *argv[ ] ) { ... }
```

Второй параметр, `argv`, является массивом указателей на символьные строки в стиле C, а первый параметр, `argc`, передает количество строк в этом массиве. Поскольку второй параметр является массивом, функцию `main()`, в качестве альтернативы, можно определить следующим образом:

```
int main( int argc, char **argv ) { ... }
```

Обратите внимание: указатель `argv` указывает на тип `char*`. При передаче аргументов функции `main()` первый элемент массива `argv` содержит либо имя программы, либо является пустой строкой. Последующие элементы передают аргументы, предоставленные в командной строке. Элемент сразу за последним указателем гарантированно будет нулем.

С учетом предыдущей командной строки `argc` содержит значение 5, а `argv` — следующие символьные строки в стиле C:

```
argv[ 0 ] = "prog"; // может также указывать на  
пустую строку  
argv[ 1 ] = "-d";  
argv[ 2 ] = "-o";  
argv[ 3 ] = "ofile";  
argv[ 4 ] = "data0";  
argv[ 5 ] = 0;
```



ВНИМАНИЕ

При использовании аргументов в массиве `argv` помните, что дополнительные аргументы начинаются с элемента `argv[1]`; элемент `argv[0]` содержит имя программы, а не введенный пользователем параметр.

Упражнения раздела 6.2.5

Упражнение 6.25. Напишите функцию `main()`, получающую два аргумента. Конкатенируйте предоставленные аргументы и выведите полученную строку.

Упражнение 6.26. Напишите программу, которая способна получать

параметры командной строки, описанные в этом разделе. Отобразите значения аргументов, переданных функции `main()`.

6.2.6. Функции с переменным количеством параметров

Иногда количество аргументов, подлежащих передаче функции, неизвестно заранее. Например, могла бы понадобиться функция, выводящая сообщения об ошибках, созданные нашей программой. Нам хотелось бы использовать одну функцию, чтобы выводить эти сообщения единообразным способом. Однако различные вызовы этой функции могли бы передавать разные аргументы, соответствующие разным видам сообщений об ошибках.

Новый стандарт предоставляет два основных способа создания функций, получающих переменное количество аргументов: если у всех аргументов тот же тип, можно передать объект библиотечного класса `initializer_list`. Если типы аргументов разные, можно написать функцию специального вида, известную как *шаблон с переменным количеством аргументов* (*variadic template*), который мы рассмотрим в разделе 16.4.

В языке C++ есть также специальный тип параметра, многоточие, применяющийся для передачи переменного количества аргументов. Мы кратко рассмотрим параметры в виде многоточия далее в этом разделе. Но следует заметить, что это средство обычно используют только в программах, которые должны взаимодействовать с функциями С.

Параметры типа `initializer_list`



Функцию, получающую произвольное количество аргументов одинакового типа, можно написать, используя параметр типа `initializer_list`. Тип `initializer_list` — это библиотечный класс, который представляет собой массив (см. раздел 3.5) значений определенного типа. Этот тип определен в заголовке `initializer_list`. Операции, предоставляемые классом `initializer_list`, перечислены в табл. 6.1.

Таблица 6.1. Операции, предоставляемые классом `initializer_list`

<code>initializer_list<T></code>	Инициализация по умолчанию; пустой список элементов типа
--	--

<code>lst;</code>	Т
<code>initializer_list<T> lst{ a, b, c... };</code>	<code>lst</code> имеет столько элементов, сколько инициализаторов; элементы являются копиями соответствующих инициализаторов. Элементы списка — константы
<code>lst2(lst) lst2 = lst</code>	Копирование или присвоение объекта класса. <code>initializer_list</code> не копирует элементы в списке. После копирования первоисточник и копия совместно используют элементы
<code>lst.size()</code>	Количество элементов в списке
<code>lst.begin() lst.end()</code>	Возвращает указатель на первый и следующий после последнего элементы <code>lst</code>

Подобно типу `vector`, тип `initializer_list` является шаблоном (см. раздел 3.3). При определении объекта класса `initializer_list` следует указать тип элементов, которые будет содержать список:

```
initializer_list<string> ls; // initializer_list
строк
initializer_list<int> li; // initializer_list
целых чисел
```

В отличие от вектора, элементы списка `initializer_list` всегда константы; нет никакого способа изменить значение его элементов.

Функцию отображения сообщений об ошибках с переменным количеством аргументов можно написать следующим образом:

```
void error_msg(initializer_list<string> il) {
    for (auto beg = il.begin(); beg != il.end(); ++beg)
        cout << *beg << " ";
    cout << endl;
}
```

Методы `begin()` и `end()` объектов класса `initializer_list` аналогичны таковым у класса `vector` (см. раздел 3.4.1). Метод `begin()` предоставляет указатель на первый элемент списка, а метод `end()` — на следующий элемент после последнего. Наша функция инициализирует переменную `beg` указателем на первый элемент и перебирает все элементы списка `initializer_list`. В теле цикла осуществляется обращение к значению `beg`, что позволяет получить доступ к текущему элементу и вывести его значение.

При передаче последовательности значений в параметре типа `initializer_list` последовательность следует заключить в фигурные скобки:

```
// expected и actual - строки
if (expected != actual)
    error_msg( "functionX", expected, actual );
else
    error_msg( "functionX", "okay" );
```

Здесь той же функции `error_msg()` передаются при первом вызове три значения, а при втором — два.

У функции с параметром `initializer_list` могут быть также и другие параметры. Например, у нашей системы отладки мог бы быть класс `ErrCode`, представляющий различные виды ошибок. Мы можем пересмотреть свою программу так, чтобы в дополнение к списку `initializer_list` передавать параметр типа `ErrCode` следующим образом:

```
void error_msg( ErrCode e, initializer_list<string>
il ) {
    cout << e.msg() << ":" ;
    for ( const auto &elem : il )
        cout << elem << " ";
    cout << endl;
}
```

Поскольку класс `initializer_list` имеет члены `begin()` и `end()`, мы можем использовать для обработки элементов серийный оператор `for` (см. раздел 5.4.3). Эта программа, как и предыдущая версия, перебирает элементы заключенного в фигурные скобки списка значений, переданных параметру `il`.

Для этой версии необходимо пересмотреть вызовы так, чтобы передать аргумент типа `ErrCode`:

```
if ( expected != actual )
    error_msg( ErrCode( 42 ), { "functionX", expected,
actual } );
else
    error_msg( ErrCode( 0 ), { "functionX", "okay" } );
```



Параметр в виде многоточия

Параметры в виде многоточия предоставляются языком C++ для взаимодействия программам с кодом на языке C, использующим такое

средство библиотеки C, как `varargs`. В других целях параметр в виде многоточия не следует использовать. Использование `varargs` описано в документации компилятора C.



Параметры в виде многоточия должны использоваться только для таких типов, которые есть и у языка C, и у C++. В частности, большинство объектов типа класса копируются неправильно, когда передаются параметру в виде многоточия.

Параметр в виде многоточия может быть только последним элементом в списке параметров и может принять любую из двух форм:

```
void foo( param_list, ... );  
void foo( ... );
```

Первая форма определяет тип (типы) для нескольких параметров функции `foo()`. Контроль типов аргументов, соответствующих определенным параметрам, осуществляется как обычно. Для аргументов, соответствующих параметру в виде многоточия, никакого контроля типов нет. В первой форме запятая после объявления параметра необязательна.

Упражнения раздела 6.2.6

Упражнение 6.27. Напишите функцию, получающую параметр типа `initializer_list<int>` и возвращающую сумму элементов списка.

Упражнение 6.28. Во второй версии функции `error_msg()`, где у нее есть параметр типа `ErrCode`, каков тип элемента в цикле `for`?

Упражнение 6.29. При использовании типа `initializer_list` в серийном операторе `for` использовали бы вы ссылку как управляющую переменную цикла? Объясните почему.

6.3. Типы возвращаемого значения и оператор `return`

Оператор `return` завершает выполнение функции и возвращает управление той функции, которая вызвала текущую. Существуют две формы оператора `return`:

```
return;  
return выражение;
```



6.3.1. Функции без возвращаемого значения

Оператор `return` без значения применим только в такой функции, типом возвращаемого значения которой объявлен `void`. Функции, возвращаемым типом которых объявлен `void`, необязательно должны содержать оператор `return`. В функции типа `void` оператор `return` неявно размещается после последнего оператора.

Как правило, функции типа `void` используют оператор `return` для преждевременного завершения выполнения. Это аналогично использованию оператора `break` (см. раздел 5.5.1) для выход из цикла. Например, можно написать функцию `swap()`, которая не делает ничего, если значения идентичны:

```
void swap( int &v1, int &v2 ) {  
    // если значения равны, их замена не нужна; можно  
    // выйти сразу  
    if ( v1 == v2 )  
        return;  
    // если мы здесь, придется поработать  
    int tmp = v2;  
    v2 = v1;  
    v1 = tmp;  
    // явно указывать оператор return не обязательно  
}
```

Сначала эта функция проверяет, не равны ли значения, и если это так, то завершает работу. Если значения не равны, функция меняет их местами. После последнего оператора присвоения осуществляется неявный выход из функции.

Функции, для возвращаемого значения которых указан тип `void`, вторую форму оператора `return` могут использовать только для возвращения результата вызова другой функции, которая возвращает тип `void`. Возвращение любого другого выражения из функции типа `void` приведет к ошибке при компиляции.



6.3.2. Функции, возвращающие значение

Вторая форма оператора `return` предназначена для возвращения результата из функции. Каждый случай возвращения значения типа, отличного от `void`, должен возвратить значение. Возвращаемое значение должно иметь тип, либо совпадающий, либо допускающий неявное преобразование (см. раздел 4.11) в тип, указанный для возвращаемого значения функции при определении.

Хотя язык C++ не может гарантировать правильность результата, он способен гарантировать, что каждое возвращаемое функцией значение будет соответствовать объявленному типу. Это может получиться не во всех случаях, компилятор попытается обеспечить возвращение значения и выход только через допустимый оператор `return`. Например:

```
// некорректное возвращение значения, этот код не
будет откомпилирован
bool str_subrange( const string &str1, const string
&str2 ) {
    // размеры одинаковы: возвратить обычный результат
    // сравнения
    if ( str1.size( ) == str2.size( ) )
        return str1 == str2; // ok: == возвращает bool
    // найти размер меньшей строки; условный оператор
    // см. раздел 4.7
    auto size = ( str1.size( ) < str2.size( ) )
                ? str1.size( ) : str2.size( );
    // просмотреть все элементы до размера меньшей
    // строки
    for ( decltype( size ) i = 0; i != size; ++i ) {
        if ( str1[ i ] != str2[ i ] )
            return; // ошибка #1: нет возвращаемого
        // значения; компилятор
        // должен обнаружить эту ошибку
    }
    // ошибка #2: выполнение может дойти до конца
    // функции, так и
```

```
// не встретив оператор return  
// компилятор может и не обнаружить эту ошибку  
}
```

Оператор `return` в цикле `for` является ошибочным потому, что он не в состоянии вернуть значение. Этую ошибку компилятор должен обнаружить.

Вторая ошибка заключается в том, что функция не имеет оператора `return` после цикла. Если произойдет вызов этой функции со строкой, являющейся подмножеством другой, процесс выполнения минует цикл `for`. Однако оператор `return` для этого случая не предусмотрен. Этую ошибку компилятор может и не обнаружить. В этом случае поведение программы во время выполнения будет непредсказуемо.



Отсутствие оператора `return` после цикла, который этот оператор содержит, является особенно коварной ошибкой. Однако большинство компиляторов ее не обнаружит.

Как возвращаются значения

Значения функций возвращаются тем же способом, каким инициализируются переменные и параметры: возвращаемое значение используется для инициализации временного объекта в точке вызова, и этот временный объект является результатом вызова функции.

В функциях, возвращающих локальные переменные, важно иметь в виду правила инициализации. Например, можно написать функцию, которой передают счетчик, слово и окончание. Функция возвращает множественную форму слова, если счетчик больше 1:

```
// возвратить множественную форму слова, если ctr  
больше 1  
string make_plural(size_t ctr, const string &word,  
                    const string &ending) {  
    return (ctr > 1) ? word + ending : word;  
}
```

Тип возвращаемого значения этой функции — `string`, это значит, что возвращаемое значение копируется в точке вызова. Функция возвращает копию значения `word` или безымянную временную строку, полученную конкатенацией `word` и `ending`.

Когда функция возвращает ссылку, она, подобно любой другой ссылке, является только другим именем для объекта, на который ссылается. Рассмотрим, например, функцию, возвращающую ссылку на более короткую из двух переданный ей строк:

```
// возвратить ссылку на строку, которая короче
const string &shorterString(const string &s1, const
string &s2) {
    return s1.size() <= s2.size() ? s1 : s2;
}
```

Параметры и возвращаемое значение имеют тип ссылки на `const string`. Строки не копируются ни при вызове функции, ни при возвращении результата.

Никогда не возвращайте ссылку на локальный объект

По завершении работы функции все хранилища ее локальных объектов освобождаются (см. раздел 6.1.1). Поэтому после завершения работы функции ссылки на ее локальные объекты ссылаются на несуществующие объекты.

```
// катастрофа: функция возвращает ссылку на
локальный объект
const string &manip() {
    string ret;
    // обработать ret некоторым образом
    if (!ret.empty())
        return ret; // ошибка: возвращение ссылки на
локальный объект!
    else
        return "Empty"; // ошибка: "Empty" - локальная
временная строка
}
```

Эта функция приведет к отказу во время выполнения, поскольку она возвращает ссылку на локальный объект. Когда функция завершит работу, область памяти, которую занимала переменная `ret`, будет освобождена. Возвращаемое значение будет ссылаться на ту область памяти, которая уже недоступна.

Оба оператора `return` возвращают здесь неопределенное значение — неизвестно, что будет, попробуй мы использовать значение, возвращенное функцией `manip()`. В первом операторе `return` очевидно, что функция пытается вернуть ссылку на локальный объект. Во втором случае

строковый литерал преобразуется в локальный временный объект класса `string`. Этот объект, как и строка `s`, является локальным объектом функции `manip()`. Область памяти, в которой располагается временный объект, освобождается по завершении функции. Оба оператора `return` возвращают ссылки на области памяти, которые больше недоступны.



Чтобы удостовериться в безопасности возвращения значения, следует задаться вопросом: к какому *существовавшему ранее* объекту относится ссылка?

Возвращать указатель на локальный объект нельзя по тем же причинам. По завершении функции локальные объекты освобождаются, и указатель указывает на несуществующий объект.

Функции, возвращающие типы класса и оператор вызова

Подобно любому оператору, оператор вызова обладает порядком (ассоциативностью) и приоритетом (см. раздел 4.1.2). У оператора вызова тот же приоритет, что и у операторов "точка" и "стрелка" (см. раздел 4.6). Как и эти операторы, оператор вызова имеет левосторонний порядок. В результате, если функция возвращает указатель, ссылку или объект типа класса, результат вызова можно использовать для обращения к члену полученного объекта.

Например, размер более короткой строки можно определить следующим образом:

```
// обращение к методу size объекта строки,
возвращенной shorterString
auto sz = shorterString(s1, s2).size();
```

Поскольку эти операторы имеют левосторонний порядок, результат вызова функции `shorterString()` будет левым операндом точечного оператора. Этот оператор выбирает метод `size()` объекта строки. Этот метод становится левым операндом второго оператора вызова.

Возвращаемая ссылка является l-значением

Является ли вызов функции l-значением (см. раздел 4.1.1), зависит от типа возвращаемого значения функции. Вызовы функций, возвращающей ссылку, являются l-значением; другие типы возвращаемого значения

являются l-значениями. Вызов функции, возвращающей ссылку, применяется таким же способом, как и любое другое l-значение. В частности, можно осуществлять присвоение результату вызова функции, возвращающей ссылку на неконстанту:

```
char &get_val(string &str, string::size_type ix) {  
    return str[ix]; // get_val подразумевает, что  
данный индекс допустим  
}  
int main() {  
    string s("a value");  
    cout << s << endl; // отображает значение  
    get_val(s, 0) = 'A'; // изменяет s[0] на A  
    cout << s << endl; // отображает значение A  
    return 0;  
}
```

Может быть несколько странно видеть вызов функции слева от оператора присвоения. Однако в этом нет ничего необычного. Возвращаемое значение — ссылка, поэтому вызов — это l-значение, а любое l-значение может быть левым операндом оператора присвоения.

Если тип возвращаемого значения является ссылкой на константу, то (как обычно) присвоение результату вызова невозможно:

```
shorterString("hi", "bye") = "X"; // ошибка:  
возвращаемое значение  
// является  
константой
```

Списочная инициализация возвращаемого значения



По новому стандарту функции могут возвращать окруженный скобками список значений. Подобно любому другому случаю возвращения значения, список используется для инициализации временного объекта, представляющего возвращение функцией значения. Если список пуст, временный объект инициализируется значением по умолчанию (см. раздел 3.3.1). В противном случае возвращаемое значение зависит от типа возвращаемого значения функции.

Для примера вернемся к функции `error_msg` из раздел 6.2.6. Эта функция получала переменное количество строковых аргументов и

выводило сообщение об ошибке, составленное из переданных строк. Теперь вместо вызова функции `error_msg()` мы возвратим вектор, содержащий строки сообщений об ошибке:

```
vector<string> process() {
    // ...
    // expected и actual - строки
    if (expected.empty())
        return {}; // возвратить пустой вектор
    else if (expected == actual)
        return {"functionX", "okay"}; // возвратить
    вектор
    //
```

инициализированный списком

```
else
    return {"functionX", expected, actual};
}
```

В первом операторе `return` возвращается пустой список. В данном случае возвращенный обработанный вектор будет пуст. В противном случае возвращается вектор, инициализированный двумя или тремя элементами, в зависимости от того, равны ли `expected` и `actual`.

У функции, возвращающей встроенный тип, заключенный в скобки список может содержать хотя бы одно значение, и это значение не должно требовать сужающего преобразования (см. раздел 2.2.1). Если функция возвращает тип класса, то используемые инициализаторы определяет сам класс (см. раздел 3.3.1).

Возвращение значения из функции `main()`

Есть одно исключение из правила, согласно которому функция с типом возвращаемого значения, отличного от `void`, обязана возвратить значение: функция `main()` может завершить работу без возвращения значения. Если процесс выполнения достигает конца функции `main()` и нет никакого значения для возвращения, компилятор неявно добавляет возвращение значения 0.

Как упоминалось в разделе 1.1, значение, возвращаемое из функции `main()`, рассматривается как индикатор состояния. Возвращение нулевого значения означает успех; большинство других значений — неудачу. У значения, отличного от нуля, есть машинно-зависимое значение. Чтобы сделать его независимым от машины, заголовок `cstdlib` определяет две

переменные препроцессора (см. раздел 2.3.2), которые можно использовать для индикации успеха или отказа:

```
int main() {
    if (some failure)
        return EXIT_FAILURE; // определено в cstdlib
    else
        return EXIT_SUCCESS; // определено в cstdlib
}
```

Поскольку это переменные препроцессора, им не должна предшествовать часть `std::` и их нельзя использовать в объявлениях `using`.

Рекурсия

Функция, которая вызывает себя прямо или косвенно, является *рекурсивной функцией* (recursive function). В качестве примера можно переписать функцию вычисления факториала так, чтобы использовать рекурсию:

```
// вычислить val!, т.е. 1 * 2 * 3 ... * val
int factorial(int val) {
    if (val > 1)
        return factorial(val-1) * val;
    return 1;
}
```

В этой реализации осуществляется рекурсивный вызов функции `factorial()`, чтобы вычислить факториал числа, начиная со значения, первоначально переданного `val`, и далее в обратном порядке. Когда значение `val` достигнет 1, рекурсия останавливается и возвращается значение 1.

В рекурсивной функции всегда должно быть определено *условие выхода* или останова (stopping condition); в противном случае рекурсия станет бесконечной, т.е. функция продолжит вызывать себя до тех пор, пока стек программы не будет исчерпан. Иногда эта ошибка называется *бесконечной рекурсией* (infinite recursion). В случае функции `factorial()` условием выхода является равенство значения параметра `val` единице.

Ниже приведена трассировка выполнения функции `factorial()` при передаче ей значения 5.

Трассировка вызова функции `factorial(5)`

Вызов	Возвращает	Значение
<code>factorial(5)</code>	<code>factorial(4) * 5</code>	120
<code>factorial(4)</code>	<code>factorial(3) * 4</code>	24
<code>factorial(3)</code>	<code>factorial(2) * 3</code>	6
<code>factorial(2)</code>	<code>factorial(1) * 2</code>	2
<code>factorial(1)</code>	1	1



Функция `main()` не может вызывать сама себя.

Упражнения раздела 6.3.2

Упражнение 6.30. Откомпилируйте версию функции `str_subrange()`, представленной в начале раздела, и посмотрите, что ваш компилятор делает с указанными сообщениями об ошибках.

Упражнение 6.31. Когда допустимо возвращение ссылки? Когда ссылки на константу?

Упражнение 6.32. Укажите, корректна ли следующая функция. Если да, то объясните, что она делает; в противном случае исправьте ошибки, а затем объясните все.

```
int &get( int *arry, int index ) { return arry[ index ]; }
int main() {
    int ia[ 10 ];
    for ( int i = 0; i != 10; ++i )
        get( ia, i ) = i;
}
```

Упражнение 6.33. Напишите рекурсивную функцию, выводящую содержимое вектора.

Упражнение 6.34. Что случится, если условие остановки функции `factorial()` будет таким:

```
if ( val != 0 )
```

Упражнение 6.35. Почему в вызове функции `factorial()` мы передали `val-1`, а не `val--`?

6.3.3. Возвращение указателя на массив

Поскольку копировать массив нельзя, функция не может возвратить его. Но функция может возвратить указатель или ссылку на массив (см. раздел 3.5.1). К сожалению, синтаксис, обычно используемый для определения функций, которые возвращают указатели или ссылки на массив, довольно сложен. К счастью, такие объявления можно упростить. Например, можно использовать псевдоним типа (см. раздел 2.5.1):

```
typedef int arrT[ 10 ]; // arrT синоним для типа
massiva из десяти
                                // целых чисел
using arrT = int[ 10 ]; // эквивалентное объявление
arrT;
                                // см. раздел 2.5.1
arrT* func( int i ); // func возвращает указатель
на массив из
                                // пяти целых чисел
```

где `arrT` — это синоним для массива из десяти целых чисел. Поскольку нельзя возвратить массив, мы определяем тип возвращаемого значения как указатель на этот тип. Таким образом, функция `func()` получает один аргумент типа `int` и возвращает указатель на массив из десяти целых чисел.

Объявление функции, возвращающей указатель на массив

Чтобы объявить функцию `func()`, не используя псевдоним типа, следует вспомнить, что размерность массива следует за определяемым именем:

```
int arr[ 10 ]; // arr массив из десяти целых
чисел
int *p1[ 10 ]; // p1 массив из десяти
указателей
int ( *p2 )[ 10 ] = &arr; // p2 указывает на массив из
десяти целых чисел
```

Подобно этим объявлениям, если необходимо определить функцию, которая возвращает указатель на массив, размерность должна следовать за именем функции. Однако функция имеет список параметров, который также следует за именем. Список параметров предшествует размерности. Следовательно, функция, которая возвращает указатель на массив, имеет такую форму:

*Тип (*функция(список_параметров)) [размерность]*

Как и в любом другом объявлении массива, *Тип* — это тип элементов, а *размерность* — это размер массива. Круглые скобки вокруг части (**функция(список_параметров)*) необходимы по той же причине, по которой они были нужны при определили указателя *p2*. Без них мы определили бы функцию, которая возвращает массив указателей.

В качестве конкретного примера рассмотрим следующее объявление функции *func()*, не использующей псевдоним типа:

```
int (*func( int i ))[ 10 ];
```

Чтобы понять это объявление, имеет смысл прочитать его следующим образом:

- *func(int)* указывает, что функцию *func()* можно вызвать с аргументом типа *int*;
- *(*func(int))* указывает, что можно обратиться к значению результата этого вызова;
- *(*func(int))[10]* указывает, что обращение к значению результата вызова функции *func()* возвращает массив из десяти элементов;
- *int (*func(int))[10]* указывает, что типом элементов этого массива является *int*.

Использование замыкающего типа возвращаемого значения



По новому стандарту есть и другой способ упростить объявления функции *func()* — с использованием *замыкающего типа возвращаемого значения* (trailing return type). Оно может быть определено для любой функции, но полезней всего оно для функций со сложными типами возвращаемого значения, такими как указатели (или ссылки) на массивы. Замыкающий тип возвращаемого значения следует за списком параметров и предваряется символом *->*. Чтобы сообщить о том, что возвращаемое значение следует за списком параметров, ключевое слово *auto* располагается там, где обычно присутствует тип возвращаемого значения:

```
// fcn получает аргумент типа int и возвращает
// указатель на массив
// из десяти целых чисел
auto func( int i ) -> int( * )[ 10 ];
```

Поскольку тип возвращаемого значения указан после списка

параметров, проще заметить, что функция `func()` возвращает указатель и что этот указатель указывает на массив из десяти целых чисел.

Использование спецификатора `decltype`



В качестве другой альтернативы, если известен массив (массивы), указатель на который способна возвратить наша функция, можно использовать спецификатор `decltype`, чтобы объявить тип возвращаемого значения. Например, следующая функция возвращает указатель на один из двух массивов, в зависимости от значения ее параметра:

```
int odd[ ] = { 1, 3, 5, 7, 9 };
int even[ ] = { 0, 2, 4, 6, 8 };
// возвращает указатель на массив из пяти элементов
типа int
decltype(odd) *arrPtr( int i ) {
    return ( i % 2 ) ? &odd : &even; // возвращает
указатель на массив
}
```

Тип возвращаемого значения функции `arrPtr()` указан как `decltype`, свидетельствуя о том, что функция возвращает указатель на любой тип, который имеет `odd`. В данном случае этот объект является массивом, поэтому функция `arrPtr()` возвращает указатель на массив из пяти целых чисел.

Единственная сложность здесь в том, что следует помнить, что спецификатор `decltype` не преобразовывает автоматически массив в указатель соответствующего ему типа. Тип, возвращенный спецификатором `decltype`, является типом массива, для которого нужно добавить `*`, чтобы указать, что функция `arrPtr()` возвращает указатель.

Упражнения раздела 6.3.3

Упражнение 6.36. Напишите объявление функции, возвращающей ссылку на массив из десяти строк, не используя ни замыкающий тип возвращаемого значения, ни спецификатор `decltype` или псевдоним типа.

Упражнение 6.37. Напишите три дополнительных объявления для функций предыдущего упражнения. Нужно использовать псевдоним типа,

замыкающий тип возвращаемого значения и спецификатор `decltype`.
Какую форму вы предпочитаете и почему?

Упражнение 6.38. Перепишите функцию `arrPtr()` так, чтобы она
возвращала ссылку на массив.



6.4. Перегруженные функции

Функции, расположенные в одной области видимости, называются *перегруженными* (overloaded), если они имеют одинаковые имена, но разные списки параметров. Пример определения нескольких функций по имени `print()` приведен в разделе 6.2.4:

```
void print( const char *cp);  
void print( const int *beg, const int *end);  
void print( const int ia[], size_t size);
```

Эти функции выполняют одинаковое действие, но их параметры относятся к разным типам. При вызове такой функции компилятор принимает решение о применении конкретной версии на основании типа переданного аргумента:

```
int j[ 2] = { 0, 1};  
print( "Hello World"); // вызов print (const  
char*)  
print(j, end(j) - begin(j)); // вызов print(const  
int*, size_t)  
print( begin(j), end(j)); // вызов print(const  
int*, const int*)
```

Перегрузка функций избавляет от необходимости придумывать (и помнить) имена, существующие только для того, чтобы помочь компилятору выяснить, которую из функций применять при вызове.



Функция `main()` не может быть перегружена.

Определение перегруженных функций

Рассмотрим приложение базы данных с несколькими функциями для поиска записи на основании имени, номера телефона, номер счета и т.д. Перегрузка функций позволит определить коллекцию функций, каждая по имени `lookup()`, которые отличаются тем, как они осуществляют поиск. Мы сможем вызвать функцию `lookup()`, передав значение любого из следующих типов:

```
Record lookup( const Account&); // поиск по счету  
Record lookup( const Phone&); // поиск по телефону  
Record lookup( const Name&); // поиск по имени  
Account acct;
```

```
Phone phone;
Record r1 = lookup( acct );           // вызов версии,
получающей Account
Record r2 = lookup( phone );          // вызов версии,
получающей Phone
```

Здесь у всех трех функций одинаковое имя, но все же это три разные функции. Чтобы выяснить, которую из них вызвать, компилятор использует тип (типы) аргументов.

Перегруженные функции должны отличаться по количеству или типу (типам) своих параметров. Каждая из функций выше получает один параметр, но типы у этих параметров разные.

Функции не могут отличаться только типами возвращаемого значения. Если списки параметров функций совпадают, а типы возвращаемого значения отличаются, то это будет ошибкой:

```
Record lookup( const Account& );
bool lookup( const Account& ); // ошибка: отличается
только типом
                                         // возвращаемого
значения
```

Различие типов параметров

Два списка параметров могут быть идентичными, даже если они не выглядят одинаково:

```
// каждая пара объявляет ту же функцию
Record lookup( const Account &acct );
Record lookup( const Account& ); // имена параметров
игнорируются
typedef Phone Telno;
Record lookup( const Phone& );
Record lookup( const Telno& );      // Telno и Phone
того же типа
```

Первое объявление в первой паре именует свой параметр. Имена параметров предназначены только для документирования. Они не изменяют список параметров.

Во второй паре типы только выглядят разными, *Telno* — не новый тип, это только синоним типа *Phone*. Псевдоним типа (см. раздел 2.5.1) предоставляет альтернативное имя для уже существующего типа, а не создает новый тип. Поэтому два параметра, отличающиеся только тем, что один использует имя типа, а другой его псевдоним, не являются разными.



Перегрузка и константные параметры

Как упоминалось в разделе 6.2.3, спецификатор `const` верхнего уровня (см. раздел 2.4.3) никак не влияет на объекты, которые могут быть переданы функции. Параметр, у которого есть спецификатор `const` верхнего уровня, неотличим от такового без спецификатора `const` верхнего уровня:

```
Record lookup( Phone );
Record lookup( const Phone ); // повторно объявляет
Record lookup( Phone )
Record lookup( Phone* );
Record lookup( Phone* const ); // повторно объявляет
                                // Record
lookup( Phone* )
```

Здесь вторые объявления повторно объявляет ту же функцию, что и первые. С другой стороны, функцию можно перегрузить на основании того, является ли параметр ссылкой (или указателем) на константную или неконстантную версию того же типа; речь идет о спецификаторе `const` нижнего уровня:

```
// функции, получающие константную и неконстантную
ссылку (или
// указатель), имеют разные параметры
Record lookup( Account& );           // функция получает
                                         // ссылку на Account
Record lookup( const Account& ); // новая функция
                                         // получает константную
                                         // ссылку
Record lookup( Account* );           // новая функция
                                         // получает указатель
                                         // на Account
Record lookup( const Account* ); // новая функция
                                         // получает указатель на
                                         // константу
```

В этих случаях компилятор может использовать константность аргумента, чтобы различить, какую функцию применять. Поскольку нет преобразования (см. раздел 4.11.2) из константы, можно передать константный объект (или указатель на константу) только версии с

константным параметром. Так как преобразование в константу возможно, можно вызвать функцию и неконстантного объекта, и указателя на неконстантный объект. Однако, как будет представлено в разделе 6.6.1, компилятор предпочтет неконстантные версии при передаче неконстантного объекта или указателя на неконстантный объект.

Совет. Когда не следует перегружать функции

Хотя перегрузка функций позволяет избежать необходимости создавать и запоминать имена общепринятых операций, она не всегда целесообразна. В некоторых случаях разные имена функций предоставляют дополнительную информацию, которая упрощает понимание программы. Давайте рассмотрим набор функций-членов класса `Screen`, отвечающих за перемещение курсора.

```
Screen& moveHome( );
Screen& moveAbs( int, int );
Screen& moveRel( int, int, string direction );
```

На первый взгляд может показаться, что этот набор функций имеет смысл перегрузить под именем `move`:

```
Screen& move( );
Screen& move( int, int );
Screen& move( int, int, string direction );
```

Однако при перегрузке этих функций мы потеряли информацию, которая была унаследована именами функции. Хотя перемещение курсора — это общая операция, совместно используемая всеми этими функциями, специфический характер перемещения уникален для каждой из этих функций. Рассмотрим, например, функцию `moveHome()`, осуществляющую вполне определенное перемещение курсора. Какое из двух приведенных ниже обращений понятнее при чтении кода?

```
// которая из записей понятней?
myScreen. moveHome( ); // вероятно, эта!
myScreen. move( );
```

Оператор `const_cast` и перегрузка

В разделе 4.11.3 упоминалось, что оператор `const_cast` особенно полезен в контексте перегруженных функций. В качестве примера вернемся к функции `shorterString()` из раздела 6.3.2:

```
// возвратить ссылку на строку, которая короче
const string &shorterString ( const string &s1,
const string &s2) {
```

```
    return s1.size() <= s2.size() ? s1 : s2;
}
```

Эта функция получает и возвращает ссылки на константную строку. Мы можем вызвать функцию с двумя неконстантными строковыми аргументами, но как результат получим ссылку на константную строку. Могла бы понадобиться версия функции `shorterString()`, которая, получив неконстантные аргументы, возвратит обычную ссылку. Мы можем написать эту версию функции, используя оператор `const_cast`:

```
string &shorterString( string &s1, string &s2) {
    auto &r = shorterString( const_cast<const string&>
(s1),
                           const_cast<const string&>
(s2));
    return const_cast<string&>( r);
}
```

Эта версия вызывает константную версию функции `shorterString()` при приведении типов ее аргументов к ссылкам на константу. Функция возвращает ссылку на тип `const string`, которая, как известно, привязана к одному из исходных, неконстантных аргументов. Следовательно, приведение этой строки назад к обычной ссылке `string&` при возвращении вполне безопасно.

Вызов перегруженной функции

Когда набор перегруженных функций определен, необходима возможность вызвать их с соответствующими аргументами. *Подбор функции* (function matching), известный также как *поиск перегруженной функции* (overload resolution), — это процесс, в ходе которого вызов функции ассоциируется с определенной версией из набора перегруженных функций. Компилятор определяет, какую именно версию функции использовать при вызове, сравнивая аргументы вызова с параметрами каждой функции в наборе.

Как правило, вовсе несложно выяснить, допустим ли вызов, и если он допустим, то какая из версий функции будет использована компилятором. Функции в наборе перегруженных версий отличаются количеством или типом аргументов. В таких случаях определить используемую функцию просто. Подбор функции усложняется в случае, когда количество параметров одинаково и они допускают преобразование (см. раздел 4.11) переданных аргументов. Распознавание вызовов компилятором при наличии преобразований рассматривается в разделе 6.6, а пока следует

понять, что при любом вызове перегруженной функции возможен один из трех результатов.

- Компилятор находит одну функцию, которая является *наилучшим соответствием* (best match) для фактических аргументов, и создает код ее вызова.
- Компилятор не может найти ни одной функции, параметры которой соответствуют аргументам вызова. В этом случае компилятор сообщает об ошибке *отсутствия соответствия* (no match).
- Компилятор находит несколько функций, которые в принципе подходят, но ни одна из них не соответствует полностью. В этом случае компилятор также сообщает об ошибке, об ошибке *неоднозначности вызова* (ambiguous call).

Упражнения раздела 6.4

Упражнение 6.39. Объясните результат второго объявления в каждом из следующих наборов. Укажите, какое из них (если есть) недопустимо.

- (a) `int calc(int, int);`
`int calc(const int, const int);`
- (b) `int get();`
`double get();`
- (c) `int *reset(int *);`
`double *reset(double *);`



6.4.1. Перегрузка и область видимости



Обычно объявлять функцию локально нежелательно. Но чтобы объяснить, как область видимости взаимодействует с перегрузкой, мы будем нарушать это правило и используем локальные объявление функции.

Новички в программировании на языке C++ зачастую не понимают взаимодействия между областью видимости и перегрузкой. Однако у перегрузки нет никаких специальных свойств относительно области видимости. Как обычно, если имя объявлено во внутренней области

видимости, оно скрывает (hidden name) такое же имя, объявленное во внешней области видимости. Имена не перегружают в областях видимости:

```
string read();
void print( const string & );
void print( double ); // перегружает функцию print
void fooBar( int ival ) {
    bool read = false; // новая область видимости:
    скрывает // предыдущее объявление имени
read
    string s = read(); // ошибка: read - переменная
типа bool, а не // функция
// плохой подход: обычно не следует объявлять
функции в локальной
// области видимости
void print( int ); // новая область видимости:
скрывает предыдущие // экземпляры функции print
print( "Value: " ); // ошибка: print( const string & )
скрыта
print( ival ); // ok: print( int ) видима
print( 3.14 ); // ok: вызов print( int );
print( double ) скрыта
}
```

Большинство читателей не удивит ошибка при вызове функции `read()`. Когда компилятор обрабатывает вызов функции `read()`, он находит локальное определение имени `read`. Это имя принадлежит переменной типа `bool`, а не функции. Следовательно, вызов некорректен.

Точно тот же процесс используется при распознавании вызова функции `print()`. Объявление `print(int)` в функции `fooBar` скрывает прежнее ее объявление. В результате будет доступна только одна функция `print()`, та, которая получает один параметр типа `int`.

Когда происходит вызов функции `print()`, компилятор ищет сначала объявление этого имени. Он находит локальное объявление функции `print()`, получающей один параметр типа `int`. Как только имя найдено, компилятор игнорирует такое же имя в любой внешней области видимости. Он полагает данное объявление единственным доступным для

использования. Остается лишь удостовериться в допустимости использования этого имени.



В языке C++ поиск имени осуществляется до проверки соответствия типов.

Первый вызов передает функции `print()` строковый литерал, но единственное ее объявление, находящееся в области видимости, имеет параметр типа `int`. Строковый литерал не может быть преобразован в тип `int`, поэтому вызов ошибочен. Функция `print(const string&)`, которая соответствовала бы этому вызову, скрыта и не рассматривается.

Когда происходит вызов функции `print()` с передачей аргумента типа `double()`, процесс повторяется. Компилятор находит локальное определение функции `print(int)`. Но аргумент типа `double` может быть преобразован в значение типа `int`, поэтому вызов корректен.

Если бы объявление `print(int)` находилось в той же области видимости, что и объявления других версий функции `print()`, это была бы еще одна ее перегруженная версия. В этом случае вызовы распознавались бы по-другому, поскольку компилятор видел бы все три функции:

```
void print(const string &);  
void print(double); // перегружает функцию print  
void print(int); // еще один экземпляр  
перегрузки  
void fooBar2(int ival) {  
    print("Value: "); // вызов print(const string &)  
    print(ival); // вызов print(int)  
    print(3.14); // вызов print(double)  
}
```

6.5. Специальные средства

В этом разделе рассматриваются три связанных с функциями средства, которые полезны во многих, но не во всех программах: аргументы по умолчанию, встраиваемые функции и функции `constexpr`, а также некоторые другие средства, обычно используемые во время отладки.

6.5.1. Аргументы по умолчанию

Параметры некоторых функций могут обладать конкретными значениями, используемыми в большинстве, но не во всех вызовах. Такие обычно используемые значения называют *аргументом по умолчанию* (default argument). Функции с аргументами по умолчанию могут быть вызваны с ними или без них.

Например, для представления содержимого окна можно было бы использовать тип `string`. Мы могли бы хотеть, чтобы по умолчанию у окна была определенная высота, ширина и фоновый символ. Но мы могли бы также захотеть позволить пользователям использовать собственные значения, кроме значений по умолчанию. Чтобы приспособить и значение по умолчанию, и определяемое пользователем, мы объявили бы функцию, представляющую окно, следующим образом:

```
typedef string::size_type sz; // typedef см. р.  
2.5.1  
string screen( sz ht = 24, sz wid = 80, char  
backgrnd = ' ' );
```

Здесь мы предоставили для каждого параметра значение по умолчанию. Аргумент по умолчанию определяется как инициализатор параметра в списке параметров. Значения по умолчанию можно определить как для одного, так и для нескольких параметров. Но если у параметра есть аргумент по умолчанию, то все параметры, следующие за ним, также должны иметь аргументы по умолчанию.

Вызов функции с аргументами по умолчанию

Если необходимо использовать аргумент по умолчанию, его значение при вызове функции пропускают. Поскольку функция `screen()` предоставляет значения по умолчанию для всех параметров, мы можем вызвать ее без аргументов, с одним, двумя или тремя аргументами:

```
string window;
```

```

window = screen( );                                // эквивалент
screen( 24, 80, ' ')
window = screen( 66);                             // эквивалент
screen( 66, 80, ' ')
window = screen( 66, 256);           // screen( 66, 256, '
')
window = screen( 66, 256, '#'); // screen( 66, 256,
'#')

```

Аргументы в вызове распознаются по позиции. Значения по умолчанию используются для аргументов, крайних справа. Например, чтобы переопределить значение по умолчанию параметра `background`, следует поставить также аргументы для параметров `height` и `width`:

```

window = screen( , , '?'); // ошибка: можно
пропустить аргументы только
                           // крайние справа
window = screen( '?' );      // вызов screen( '?', 80,
' ')

```

Обратите внимание, что второй вызов, передающий одно символьное значение, вполне допустим. Несмотря на допустимость, это вряд ли то, что ожидалось. Вызов допустим потому, что символ '`?`' имеет тип `char`, а он может быть преобразован в тип крайнего левого параметра. Это параметр типа `string::size_type`, который является целочисленным беззнаковым типом. В этом вызове аргумент типа `char` неявно преобразуется в тип `string::size_type` и передается как аргумент параметру `height`. На машине авторов символ '`?`' имеет шестнадцатеричное значение `0x3F`, соответствующее десятичному `63`. Таким образом, этот вызов присваивает параметру `height` значение `63`.

Одной из задач при разработке функции с аргументами по умолчанию является упорядочивание параметров так, чтобы те из них, для которых использование значения по умолчанию вероятней всего, располагались последними.

Объявление аргумента по умолчанию

Хотя вполне обычной практикой является объявление функции однажды в заголовке, вполне допустимо многократно объявлять ее повторно. Однако у каждого параметра может быть свое значение по умолчанию, определенное только однажды в данной области видимости. Таким образом, любое последующее объявление может добавить значение

по умолчанию только для того параметра, у которого ранее не было определено значение по умолчанию. Как обычно, значения по умолчанию могут быть определены, только если у всех параметров справа уже есть значения по умолчанию. Рассмотрим следующий пример:

```
// у параметров height и width нет значений по умолчанию
```

```
string screen( sz, sz, char = ' ' );
```

Нельзя изменить уже заявленное значение по умолчанию:

```
string screen( sz, sz, char = '*' ); // ошибка:  
переобъявление
```

Но можно добавить аргумент по умолчанию следующим образом:

```
string screen( sz = 24, sz = 80, char ); // ok:  
добавление аргументов
```

```
// по
```

умолчанию



Обычно аргументы по умолчанию определяют при объявлении функции в соответствующем заголовке.

Инициализация аргумента по умолчанию

Локальные переменные не могут использоваться как аргумент по умолчанию. За исключением этого ограничения, аргумент по умолчанию может быть любым выражением, тип которого приводим к типу параметра:

```
// объявления wd, def и ht должны располагаться вне  
функции
```

```
sz wd = 80;  
char def = ' ' ;  
sz ht();  
string screen( sz = ht(), sz = wd, char = def );  
string window = screen(); // вызов screen( ht(), 80,  
' ' )
```

Поиск имен, используемых для аргументов по умолчанию, осуществляется в пределах объявления функции. Значения, представляемые этими именами, вычисляются во время вызова:

```
void f2() {  
    def = '*' ; // изменение значения аргумента по
```

умолчанию

```
sz wd = 100; // скрывает внешнее определение wd,  
но не изменяет  
        // значение по умолчанию  
window = screen(); // вызов screen( ht(), 80, '*' )  
}
```

В функции `f2()` было изменено значение `def`. Вызов функции `screen` передает это измененное значение. Эта функция также объявляет локальную переменную, которая скрывает внешнюю переменную `wd`. Однако локальное имя `wd` никак не связано с аргументом по умолчанию, переданным функции `screen()`.

Упражнения раздела 6.5.1

Упражнение 6.40. Какое из следующих объявлений (если оно есть) содержит ошибку? Почему?

- (a) `int ff(int a, int b = 0, int c = 0);`
- (b) `char *init(int ht = 24, int wd, char bckgrnd);`

Упражнение 6.41. Какие из следующих вызовов (если они есть) недопустимы? Почему? Какие из них допустимы (если они есть), но, вероятно, не соответствуют намерениям разработчика? Почему?

```
char *init( int ht, int wd = 80, char bckgrnd = '  
' );  
( a ) init(); ( b ) init( 24, 10 ); ( c ) init( 14, '*' );
```

Упражнение 6.42. Присвойте второму параметру функции `make_plural()` (см. раздел 6.3.2) аргумент по умолчанию '`s`'. Проверьте программу, выведя слова "success" и "failure" в единственном и множественном числе.

6.5.2. Встраиваемые функции и функции `constexpr`

В разделе 6.3.2 приведена небольшая функция, возвращающая ссылку на более короткую строку из двух переданных ей. К преимуществам определения функции для такой маленькой операции относятся следующие.

- Обращение к функции `shorterString()` проще и понятнее, чем эквивалентное условное выражение.
- Использование функции гарантирует одинаковое поведение. Она гарантирует, что каждая проверка будет выполнена тем же способом.
- Если придется внести изменение, проще сделать это в теле функции, а

не выискивать в коде программы все случаи применения эквивалентного выражения.

- Функция может быть многократно использована при написании других приложений.

Однако у функции `shorterString()` есть один потенциальный недостаток: ее вызов происходит медленнее, чем вычисление эквивалентного выражения. На большинстве машин при вызове функции осуществляется довольно много действий: перед обращением сохраняются регистры, которые необходимо будет восстановить после выхода; происходит копирование значений аргументов; управление программой переходит к новому участку кода.

Встраиваемые функции позволяют избежать дополнительных затрат на вызов

Содержимое функции, объявленной *встраиваемой* (`inline`) при компиляции, как правило, встраивается по месту вызова. Предположим, что функция `shorterString()` объявлена встраиваемой, а ее вызов имеет такой вид:

```
cout << shorterString( s1, s2 ) << endl;
```

При компиляции тело функции окажется встроено по месту вызова, и в результате получится нечто вроде следующего:

```
cout << ( s1.size( ) < s2.size( ) ? s1 : s2 ) << endl;
```

Таким образом, во время выполнения удастся избежать дополнительных затрат, связанных с вызовом функции `shorterString()`.

Чтобы объявить функцию `shorterString()` встраиваемой, в определении, перед типом возвращаемого значения, располагают ключевое слово `inline`.

```
// встраиваемая версия функции сравнения двух строк
inline const string &
shorterString( const string &s1, const string &s2 ) {
    return s1.size( ) <= s2.size( ) ? s1 : s2;
}
```



Объявление функции встраиваемой является только *рекомендацией* компилятору. Компилятор вполне может проигнорировать эту

рекомендацию.

На самом деле механизм встраивания применяется в процессе оптимизации объектного кода, в ходе которого код небольших функций, вызов которых происходит достаточно часто, встраивается по месту вызова. Большинство компиляторов не будет встраивать рекурсивные функции. Функция на 75 строк также, вероятно, не будет встроена.

Функции `constexpr`



Функция `constexpr` — это функция, которая может быть применена в константном выражении (см. раздел 2.4.4). Функция `constexpr` определяется как любая другая функция, но должна соответствовать определенным ограничениям: возвращаемый тип и тип каждого параметра должны быть литералами (см. раздел 2.4.4), тело функции должно содержать только один оператор `return`:

```
constexpr int new_sz() { return 42; }
constexpr int foo = new_sz(); // ok: foo -  
константное выражение
```

Здесь функция `new_sz` определена как `constexpr`, она не получает никаких аргументов. Компилятор может проверить (во время компиляции), что вызов функции `new_sz()` возвращает константное выражение, поэтому ее можно использовать для инициализации переменной `constexpr` по имени `foo`.

Если это возможно, компилятор заменит вызов функции `constexpr` ее результирующим значением. Для этого функция `constexpr` неявно считается встраиваемой.

Тело функции `constexpr` может содержать другие операторы, если они не выполняют действий во время выполнения. Например, функция `constexpr` может содержать пустые операторы, псевдонимы типа (см. раздел 2.5.1) и объявления `using`.

Функции `constexpr` позволено возвратить значение, которое не является константой:

```
// scale(arg) - константное выражение, если arg -
константное выражение
constexpr size_t scale(size_t cnt) { return
new_sz() * cnt; }
```

Функция `scale()` возвратит константное выражение, если ее аргумент будет константным выражением, но не в противном случае:

```
int arr[ scale( 2 ) ]; // ok: scale( 2 ) - константное
выражение
int i = 2;           // i - неконстантное выражение
int a2[ scale( i ) ]; // ошибка: scale( i ) -
неконстантное выражение
```

Если передать константное выражение (такое как литерал `2`), возвращается тоже константное выражение. В данном случае компилятор заменит вызов функции `scale()` результирующим значением.

Если происходит вызов функции `scale()` с выражением, которое не является константным (например, объект `i` типа `int`), то возвращается неконстантное выражение. Если использовать функцию `scale()` в контексте, требующем константного выражения, компилятор проверит, является ли результат константным выражением. Если это не так, то компилятор выдаст сообщение об ошибке.

Функция `constexpr` не обязана возвращать константное выражение.

Помещайте встраиваемые функции и функции `constexpr` в файлы заголовка

В отличие от других функций, встраиваемые функции и функции `constexpr` могут быть определены в программе несколько раз. В конце концов, чтобы встроить код, компилятор нуждается в определении, а не только в объявлении. Однако все определения конкретной встраиваемой функции и функции `constexpr` должны совпадать точно. В результате встраиваемые функции и функции `constexpr` обычно определяют в заголовках.

Упражнения раздела 6.5.2

Упражнение 6.43. Какое из следующих объявлений и определений имеет смысл поместить в файл заголовка, а какой — в текст файла исходного кода? Объясните почему.

- (a) `inline bool eq(const BigInt&, const BigInt&) . . . }`
- (b) `void putValues(int *arr, int size);`

Упражнение 6.44. Перепишите функцию `isShorter()` из раздела 6.2.2 как встраиваемую.

Упражнение 6.45. Пересмотрите функции, написанные для

предыдущих упражнений, и решите, должны ли они быть определены как встраиваемые. Если да, то сделайте это. В противном случае объясните, почему они не должны быть встраиваемыми.

Упражнение 6.46. Возможно ли определить функцию `isShorter` как `constexpr`? Если да, то сделайте это. В противном случае объясните, почему нет.

6.5.3. Помощь в отладке

Для условного выполнения отладочного кода программисты C++ иногда используют подход, подобный защите заголовка (см. раздел 2.6.3). Идея в том, что программа будет содержать отладочный код, который выполняется только во время разработки программы. Когда приложение закончено и готово к выпуску, отладочный код исключается. Этот подход подразумевает использование двух средств препроцессора: `assert` и `NDEBUG`.

Макрос препроцессора assert

Макрос assert — это макрос препроцессора (preprocessor macro). Макрос препроцессора — это переменная препроцессора, действующая как встраиваемая функция. Макрос `assert` получает одно выражение и использует его как условие:

```
assert( выражение );
```

Если результат выражения ложь (т.е. нуль), то макрос `assert` выдает сообщение и закрывает программу. Если результат выражения — истина (т.е. он отличен от нуля), то макрос `assert` не делает ничего.

Действие макроса препроцессора подобно вызову функции. Макрос `assert` получает одно выражение, которое он использует как условие.

Макрос `assert` определен в заголовке `cassert`. Как уже упоминалось, относящиеся к препроцессору имена обрабатывает препроцессор, а не компилятор (см. раздел 2.3.2). В результате такие имена можно использовать непосредственно, без объявления `using`. Таким образом, используется имя `assert`, а не `std::assert`, кроме того, для него не предоставляется объявление `using`.

Макрос `assert` зачастую используется для проверки "недопустимых" условий. Например, программа обработки вводимого текста могла бы проверять, что все вводимые слова длиннее некоего порогового значения. Эта программа могла бы содержать такой оператор:

```
assert( word.size() > threshold);
```

Переменная препроцессора NDEBUG

Поведение макроса assert зависит от состояния переменной препроцессора NDEBUG. Если переменная NDEBUG определена, макрос assert ничего не делает. По умолчанию переменная NDEBUG не определена, поэтому по умолчанию макрос assert выполняет проверку.

Отладку можно "выключить", предоставив директиву #define, определяющую переменную NDEBUG. В качестве альтернативы большинство компиляторов предоставляет параметр командной строки, позволяющий определять переменные препроцессора:

```
$ CC -D NDEBUG main.C # use /D with the Microsoft compiler
```

Результат будет тот же, что и при наличии строки #define NDEBUG в начале файла main.C.

Когда переменная NDEBUG определена, программа во время выполнения избегает дополнительных затрат на проверку различных условий. Самих проверок во время выполнения, конечно, тоже не будет. Поэтому макрос assert следует использовать только для проверки того, что действительно недопустимо. Это может быть полезно при отладке программы, но не должно использоваться для замены логических проверок времени выполнения или проверки ошибок, которые должна осуществлять программа.

В дополнение к макросу assert можно написать собственный отладочный код, выполняющийся в зависимости от переменной NDEBUG. Если переменная NDEBUG не определена, код между директивами #ifndef и#endif выполняется, а в противном случае игнорируется:

```
void print( const int ia[], size_t size) {  
    #ifndef NDEBUG  
    // __func__ - локальная статическая переменная,  
    // определенная  
    // компилятором. Она содержит имя функции  
    cerr << __func__ << ": array size is " << size <<  
    endl;  
    #endif  
    // ...
```

Здесь переменная __func__ используется для вывода имени отлаживаемой функции. Компилятор определяет переменную __func__ в

каждой функции. Это локальный статический массив типа `const char`, содержащий имя функции.

Кроме переменной `_func_`, определяемой компилятором C++, препроцессор определяет четыре других имени, которые также могут пригодиться при отладке:

`_FILE_` строковый литерал, содержащий имя файла.

`_LINE_` целочисленный литерал, содержащий номер текущей строки.

`_TIME_` строковый литерал, содержащий файл и время компиляции.

`_DATE_` строковый литерал, содержащий файл и дату компиляции.

Эти константы можно использовать для отображения дополнительной информации в сообщениях об ошибках:

```
if (word.size() < threshold)
    cerr << "Error: " << _FILE_
        << " : in function " << _func_
        << " at line " << _LINE_ << endl
        << " Compiled on " << _DATE_
        << " at " << _TIME_ << endl
        << " Word read was \" " << word << "\": Length
too short" << endl;
```

Если передать этой программе строку, которая короче `threshold`, то будет создано следующее сообщение об ошибке:

```
Error: wdebug.cc : in function main at line 27
Compiled on Jul 11 2012 at 20:50:03
Word read was "foo": Length too short
```

Упражнения раздела 6.5.3

Упражнение 6.47. Пересмотрите программу, написанную в упражнении раздела 6.3.2, где использовалась рекурсия для отображения содержимого вектора так, чтобы условно отображать информацию о ее выполнении. Например, отобразите размер вектора при каждом вызове. Откомпилируйте и запустите программу с включенной отладкой и с выключенными.

Упражнение 6.48. Объясните, что делает этот цикл и стоит ли использовать в нем макрос `assert`:

```
string s;
while (cin >> s && s != sought) { } // пустое тело
assert(cin);
```



6.6. Подбор функции

Во многих (если не во всех) случаях довольно просто выяснить, какая из перегруженных версий функции будет использована при данном вызове. Но это не так просто, когда у перегруженных функций одинаковое количество параметров и когда один или несколько параметров имеют типы, связанные преобразованиями. Для примера рассмотрим следующий набор перегруженных функций и их вызов:

```
void f() ;  
void f( int ) ;  
void f( int, int ) ;  
void f( double, double = 3.14 ) ;  
f( 5.6 ); // вызов void f( double, double )
```

Выявление кандидатов и подходящих функций

На первом этапе подбора перегруженной функции выявляют набор версий, подходящих для рассматриваемого вызова. Такие функции называются *функциями-кандидатами* (candidate function). Функция-кандидат имеет имя, указанное при вызове, и видима в точке вызова. В данном примере кандидатами являются все четыре функции по имени *f*.

На втором этапе выбора функции из набора кандидатов выявляются те, которые могут быть вызваны с аргументами данного вызова. Выбранные функции называют *подходящими* (viable function). Чтобы считаться подходящей, функция должна иметь столько же параметров, сколько аргументов передано при вызове, и тип каждого аргумента должен совпадать или допускать преобразование в тип соответствующего параметра.

При вызове *f(5.6)* две функции-кандидата можно исключить сразу из-за несоответствия количеству аргументов. Речь идет о версии без параметров и версии с двумя параметрами типа *int*. В данном случае вызов имеет только один аргумент, а эти функции не имеют их вообще или имеют два параметра соответственно.

Функция, получающая один аргумент типа *int*, и функция, получающая два аргумента типа *double*, могли бы быть подходящими. Любая из них может быть вызвана с одним аргументом. Функция, получающая два аргумента типа *double*, имеет аргумент по умолчанию, а

значит, может быть вызвана с одним аргументом.



Когда у функции есть аргументы по умолчанию (см. раздел 6.5.1), при вызове может быть передано меньше аргументов, чем она фактически имеет.

После проверки количества аргументов, позволяющей выявить функции, подходящие потенциально, проверяется соответствие типов параметров функций типам аргументов, переданных при вызове. Как и при любом обращении, тип аргумента может либо совпадать, либо допускать преобразование в тип параметра. В данном случае подходят обе оставшиеся функции.

- Функция `f(int)` является подходящей потому, что аргумент типа `double` может быть неявно преобразован в параметр типа `int`.
- Функция `f(double, double)` также является подходящей потому, что для второго параметра задано значение по умолчанию, а первый параметр имеет тип `double`, который точно соответствует типу аргумента.



Если никаких подходящих функций не обнаружено, компилятор выдает сообщение об ошибке.

Поиск наилучшего соответствия, если он есть

На третьем этапе подбора перегруженной функции выясняется, какая из допустимых функций наилучшим образом соответствует вызову. Этот процесс анализирует каждый аргумент вызова и выбирает подходящую функцию (или функции), для которой соответствие параметра аргументу является наилучшим. Подробно критерии наилучшего соответствия рассматриваются в следующем разделе, а пока достаточно знать, что чем ближе типы аргумента и параметра друг к другу, тем лучше соответствие.

В данном случае существует только один (явный) аргумент, который имеет тип `double`. При вызове версии `f(int)` аргумент преобразуется из типа `double` в тип `int`. Вторая подходящая функция, `f(double,`

`double`) , точно соответствует типу этого аргумента. Поскольку точное соответствие лучше соответствия требующего преобразования, компилятор предпочитает версию с двумя параметрами типа `double`. Для второго, недостающего аргумента компилятор добавит аргумент по умолчанию.

Подбор перегруженной версии с несколькими параметрами

Если у функции два или несколько аргументов, подбор подходящей версии усложняется. Предположим, что функции имеют то же имя `f`, но анализируется следующий вызов:

`f(42, 2.56);`

Набор подходящих функций выявляется, как прежде. Компилятор выбирает те версии функции, которые имеют необходимое количество параметров, типы которых соответствуют типам аргументов. В данном случае в набор подходящих вошли функции `f(int, int)` и `f(double, double)`. Затем компилятор перебирает аргументы один за одним и определяет, какая из версий функций имеет наилучшее соответствие. Наилучше соответствующая функция та, для которой единственной выполняются следующие условия.

- Соответствие по каждому аргументу не хуже, чем у остальных подходящих функций.
- По крайней мере у одного аргумента соответствие лучше, чем у остальных подходящих функций.

Если после просмотра всех аргументов не было найдено ни одной функции, которая считалась бы наилучше соответствующей, компилятор сообщает об ошибке неоднозначности вызова.

В рассматриваемом примере вызова анализ лишь первого аргумента для версии `f(int, int)` функции `f()` обнаруживает точное соответствие. При анализе второй версии функции `f()` оказывается, что аргумент 42 типа `int` следует преобразовать в значение типа `double`. Соответствие в результате встроенного преобразования хуже, чем точное. Таким образом, рассматривая только этот параметр, лучше соответствует та версия функции `f()`, которая обладает двумя параметрами типа `int`, а не двумя параметрами типа `double`.

Но при переходе ко второму аргументу оказывается, что версия функции `f()` с двумя параметрами типа `double` точно соответствует аргументу 2.56. Вызов версии функции `f()` с двумя параметрами типа `int` потребует преобразования аргумента 2.56 из типа `double` в тип `int`. Таким образом, при рассмотрении только второго параметра версия

`f(double, double)` функции `f()` имеет лучшее соответствие.

Компилятор отклонит этот вызов, поскольку он неоднозначен: каждая подходящая функция является лучшим соответствием по одному из аргументов. Было бы заманчиво обеспечить соответствие за счет явного приведения типов (см. раздел 4.11.3) одного из аргументов. Но в хорошо спроектированных системах в приведении аргументов не должно быть необходимости.



Рекомендуем

При вызове перегруженных функций приведения аргументов практически не нужны: потребность в приведении означает, что наборы параметров перегруженных функций проработаны плохо.

Упражнения раздела 6.6

Упражнение 6.49. Что такое функция-кандидат? Что такое подходящая функция?

Упражнение 6.50. С учетом приведенных в начале раздела объявлений функции `f()` перечислите подходящие функции для каждого из следующих вызовов. Укажите наилучше соответствие, или если его нет, то из-за отсутствия соответствия или неоднозначности вызова?

- (a) `f(2. 56, 42)` (b) `f(42)` (c) `f(42, 0)` (d) `f(2. 56, 3. 14)`

Упражнение 6.51. Напишите все четыре версии функции `f()`. Каждая из них должна выводить собственное сообщение. Проверьте свои ответы на предыдущее упражнение. Если ответы были неправильными, перечитайте этот раздел и выясните, почему вы ошиблись.



6.6.1. Преобразование типов аргументов

Чтобы определить наилучшее соответствие, компилятор ранжирует преобразования, применяемые для приведения типа аргумента к типу соответствующего ему параметра. Преобразования ранжируются в порядке убывания следующим образом.

1. Точное соответствие. Типы аргумента и параметра совпадают в случае, если:

- типы аргумента и параметра идентичны;
 - аргумент преобразуется из типа массива или функции в соответствующий тип указателя. (Указатели на функции рассматриваются в разделе 6.7);
 - аргумент отличается наличием или отсутствием спецификатора `const` верхнего уровня.
2. Соответствие в результате преобразования констант (см. раздел 4.11.2).
 3. Соответствие в результате преобразования (см. раздел 4.11.1).
 4. Соответствие в результате арифметического преобразования (см. раздел 4.11.1) или преобразования указателя (см. раздел 4.11.2).
 5. Соответствие в результате преобразования класса (раздел 14.9).



Соответствие, требующее приведения и (или) целочисленного преобразования



ВНИМАНИЕ

В контексте соответствия функций приведение и преобразование встроенных типов может привести к удивительным результатам. К счастью, в хорошо разработанных системах редко используют функции с параметрами, столь похожими, как в следующих примерах.

При анализе вызова следует помнить, что малые целочисленные типы всегда преобразуются в тип `int` или больший целочисленный тип. Рассмотрим две функции, одна из которых получает тип `int`, а вторая тип `short`, версия `short` будет вызвана только со значениями типа `short`. Даже при том, что меньшие целочисленные значения могли бы быть ближе к соответствию, эти значения преобразуются в тип `int`, тогда как вызов версии `short` потребовал бы преобразования:

```
void ff( int );
void ff( short );
ff('a'); // тип char приводится к int, поэтому
применяется f( int )
```

Все целочисленные преобразования считаются эквивалентными друг другу. Преобразование из типа `int` в `unsigned int`, например, не имеет

преимущества перед преобразованием типа `int` в `double`. Рассмотрим конкретный пример.

```
void manip( long );
void manip( float );
manip( 3.14 ); // ошибка: неоднозначный вызов
```

Литерал 3.14 имеет тип `double`. Этот тип может быть преобразован или в тип `long`, или в тип `float`. Поскольку возможны два целочисленных преобразования, вызов неоднозначен.

Соответствие функций и константные аргументы

Когда происходит вызов перегруженной функции, различие между версиями которой заключается в том, указывает ли параметр (или ссылается) на константу, компилятор способен различать, является ли аргумент константным или нет:

```
Record lookup( Account& ); // функция,
получающая ссылку на Account
Record lookup( const Account& ); // новая функция,
получающая ссылку на
// константу
const Account a;
Account b;
lookup( a ); // вызов lookup( const Account& )
lookup( b ); // вызов lookup( Account& )
```

В первом вызове передается константный объект `a`. Нельзя связать простую ссылку с константным объектом. В данном случае единственная подходящая функция — версия, получающая ссылку на константу. Кроме того, этот вызов точно соответствует аргументу `a`.

Во втором вызове передается неконстантный объект `b`. Для этого вызова подходят обе функции. Аргумент `b` можно использовать для инициализации ссылки константного или неконстантного типа. Но инициализация ссылки на константу неконстантным объектом требует преобразования. Версия, получающая неконстантный параметр, является точным соответствием для объекта `b`. Следовательно, неконстантная версия предпочтительней.

Параметры в виде указателя работают подобным образом. Если две функции отличаются только тем, указывает ли параметр на константу или не константу, компилятор на основании константности аргумента вполне может решить, какую версию функции использовать: если аргумент

является указателем на константу, то вызов будет соответствовать версии, получающей тип `const*`; в противном случае, если аргумент — указатель на не константу, вызывается версия, получающая простой указатель.

Упражнения раздела 6.6.1

Упражнение 6.52. Предположим, что существуют следующие объявления:

```
void manip( int, int );
double dobj;
```

Каков порядок (см. раздел 6.6.1) преобразований в каждом из следующих обращений?

(a) `manip('a' , 'z') ;` (b) `manip(55.4 , dobj) ;`

Упражнение 6.53. Объясните назначение второго объявления в каждом из следующих наборов. Укажите, какие из них (если они есть) недопустимы.

(a) `int calc(int& , int&);`
`int calc(const int& , const int&);`

(b) `int calc(char* , char*);`
`int calc(const char* , const char*);`

(c) `int calc(char* , char*);`
`int calc(char* const , char* const);`

6.7. Указатели на функции

Указатель на функцию (function pointer) содержит адрес функции, а не объекта. Подобно любому другому указателю, указатель на функцию имеет вполне определенный тип. Тип функции определен типом ее возвращаемого значения и списком параметров. Имя функции не является частью ее типа.

```
// сравнивает длины двух строк
bool lengthCompare( const string &, const string &);
```

Эта функция имеет тип `bool(const string&, const string&)`. Чтобы объявить указатель, способный указывать на эту функцию, достаточно расположить указатель вместо имени функции:

```
// rf указывает на функцию, получающую две
константные ссылки
// на строки и возвращающую значение типа bool
bool ( *rf )( const string &, const string & ); // не
инициализирован
```

Просматривая объявление с начала, можно заметить, что имени `rf` предшествует знак `*`, следовательно, `rf` — указатель. Справа расположен список параметров, означая, что `rf` указывает на функцию. Глядя влево, можно заметить, что возвращаемым типом функции является `bool`. Таким образом, указатель `rf` указывает на функцию, которая имеет два параметра типа `const string&` и возвращает значение типа `bool`.



Круглые скобки вокруг части `* rf` необходимы. Без них получится объявление функции `rf()`, возвращающей указатель на тип `bool`:

```
// объявление функции rf(), возвращающей указатель
на тип bool
bool *rf( const string &, const string & );
```

Использование указателей на функцию

При использовании имени функции как значения функция автоматически преобразуется в указатель. Например, адрес функции `lengthCompare()` можно присвоить указателю `rf` следующим образом:

```

    pf = lengthCompare;      // pf теперь указывает на
функцию lengthCompare
    pf = &lengthCompare; // эквивалентное присвоение:
оператор обращения к
                                // адресу необязателен

```

Кроме того, указатель на функцию можно использовать для вызова функции, на которую он указывает. Это можно сделать непосредственно, обращение к значению указателя там не обязательно:

```

bool b1 = pf("hello", "goodbye");           //
вызов lengthCompare
bool b2 = (*pf)("hello", "goodbye");         //
эквивалентный вызов
bool b3 = lengthCompare("hello", "goodbye");   //
эквивалентный вызов

```

Преобразование указателя на один тип функции в указатель на другой тип функции невозможно. Однако для обозначения того, что указатель не указывает на функцию, ему можно присвоить nullptr (см. раздел 2.3.2) или целочисленное константное выражение, означающее нуль:

```

string::size_type sumLength(const string&, const
string&);

bool cstringCompare(const char*, const char*); // ok: pf не указывает на
функцию
pf = 0;                                         // ошибка: разные типы
возвращаемого значения
pf = cstringCompare; // ошибка: разные типы
параметров
pf = lengthCompare; // ok: типы функции и
указателя совпадают точно

```

Указатели на перегруженные функции

Как обычно, при использовании перегруженной функции применяемую версию должен прояснить контекст, в котором она используется. Вот объявление указателя на перегруженную функцию:

```

void ff(int*);
void ff(unsigned int);
void (*pf1)(unsigned int) = ff; // pf1 указывает на
ff(unsigned)

```

Компилятор использует тип указателя для выявления используемой

версии перегруженной функции. Тип указателя должен точно соответствовать одной из версий перегруженной функции:

```
void (*pf2)(int) = ff;           // ошибка: нет версии с
только таким списком
                                         // параметров
double (*pf3)(int*) = ff;        // ошибка: тип
возвращаемого значения
                                         // функций ff и pf3 не
совпадают
```

Указатель на функцию как параметр

Подобно массивам (см. раздел 6.2.4), нельзя определить параметры типа функции, но можно создать параметр, являющийся указателем на функцию. Как и в случае с массивами, можно создать параметр, который выглядит как тип функции, но обрабатывается как указатель:

```
// третий параметр имеет тип функции и
автоматически обрабатывается как
// указатель на функцию
void useBigger(const string &s1, const string &s2,
               bool pf(const string&, const
string&));
// эквивалентное объявление: параметр явно
определен как указатель
// на функцию
void useBigger(const string &s1, const string &s2,
               bool (*pf)(const string&, const
string&));
```

При передаче функции как аргумента это можно сделать непосредственно. Аргумент будет автоматически преобразован в указатель:

```
// автоматическое преобразование функции
lengthCompare в указатель
// на нее
useBigger(s1, s2, lengthCompare);
```

Как можно заметить в объявлении функции `useBigger()`, написание указателей на тип функций быстро становится утомительным. Псевдонимы типа (см. раздел 2.5.1), а также спецификатор `decltype` (см. раздел 2.5.3) позволяют упростить код, который использует указатели на функции:

```
// Func и Func2 имеют тип функции
```

```

typedef bool Func( const string&, const strings);
typedef decltype( lengthCompare) Func2;      // Эквивалентный тип
// FuncP и FuncP2 имеют тип указателя на функцию
typedef bool( *FuncP)( const string&, const string&);
typedef decltype( lengthCompare) *FuncP2;      // Эквивалентный тип

```

Здесь при определении типов использовано ключевое слово `typedef`. И `Func`, и `Func2` являются типами функций, тогда как `FuncP` и `FuncP2` — типы указателя. Следует заметить, что спецификатор `decltype` возвращает тип функции; автоматического преобразования в указатель не происходит. Поскольку спецификатор `decltype` возвращает тип функции, при необходимости получить указатель следует добавить символ `*`. Можно повторно объявить функцию `useBigger()`, используя любой из этих типов:

```

// Эквивалентные объявления useBigger с
использованием псевдонимов типа
void useBigger( const string&, const string&, Func);
void useBigger( const string&, const string&, FuncP2);

```

Оба объявления объявляют ту же функцию. В первом случае компилятор автоматически преобразует тип функции, представленный именем `Func`, в указатель.

Возвращение указателя на функцию

Подобно массивам (см. раздел 6.3.3), нельзя возвратить тип функции, но можно возвратить указатель на тип функции. Точно так же тип возвращаемого значения следует писать как тип указателя; компилятор не будет автоматически рассматривать тип возвращаемого значения функции как соответствующий тип указателя. Как и при возвращении массива, безусловно, проще всего объявить функцию, которая возвращает указатель на функцию, при помощи псевдонима типа:

```

using F = int( int*, int);      // F - тип функции, а
не указатель
using PF = int( * )( int*, int); // PF - тип указателя

```

Здесь для определения `F` как типа функции и `PF` как указателя на тип функции было использовано объявление псевдонима типа (см. раздел 2.5.1). Имейте в виду, что в отличие от параметров, имеющих тип функции,

типа возвращаемого значения не преобразуется автоматически в тип указателя. Следует явно определить, что тип возвращаемого значения является типом указателя:

```
PF f1(int); // ok: PF - указатель на функцию; f1
возвращает указатель
                                // на функцию
F f1(int); // ошибка: F - тип функции; f1 не может
возвратить функцию
F *f1(int); // ok: явное определение типа
возвращаемого значения как
                                // указателя на функцию
```

Конечно, функцию `f1()` также можно объявить непосредственно:

```
int (*f1(int))(int*, int);
```

Читая это объявление изнутри наружу, можно заметить у `f1` список параметров, таким образом, `f1` — это функция. Имени `f1` предшествует знак `*`, следовательно, функция `f1()` возвращает указатель. У типа самого указателя тоже есть список параметров, таким образом, указатель указывает на функцию. Эта функция возвращает тип `int`.

И наконец, следует обратить внимание на то, что объявления функций, которые возвращают указатель на функцию, можно упростить при помощи замыкающего типа возвращаемого значения (см. раздел 6.3.3):

```
auto f1(int) -> int (*)(int*, int);
```

Использование спецификаторов `auto` и `decltype` для типов указателей на функции

Если известно, какую функцию (функции) следует возвратить, можно использовать спецификатор `decltype` для упрощения записи типа возвращаемого значения в виде указателя на функцию. Предположим, например, что имеются две функции, обе возвращают тип `string::size_type` и имеют два параметра типа `const string&`. Можно написать третью функцию, которая получает параметр типа `string` и возвращает указатель на одну из следующих двух функций следующим образом:

```
string::size_type sumLength(const string&, const
string&);
string::size_type largerLength(const string&, const
string&);
// в зависимости от значения строкового параметра
```

```
функция getFcn
    // возвращает указатель на sumLength или
largerLength
    decltype( sumLength ) *getFcn( const string & );
```

Единственная сложность в объявлении функции `getFcn()` — это необходимость помнить, что при применении спецификатора `decltype` к функции она возвращает тип функции, а не указатель на тип функции. Чтобы получить указатель, а не функцию, следует добавить знак `*`.

Упражнения раздела 6.7

Упражнение 6.54. Напишите объявление функции, получающей два параметра типа `int`, и возвращающей тип `int`. Объявите также вектор, элементами которого является тип указателя на эту функцию.

Упражнение 6.55. Напишите четыре функции, которые добавляют, вычитают, умножают и делят два значения типа `int`. Сохраните указатели на эти значения в векторе из предыдущего упражнения.

Упражнение 6.56. Обратитесь к каждому элементу вектора и выведите результат.

Резюме

Функции представляют собой именованные блоки действий, применяемые для структурирования даже небольших программ. При их определении указывают тип возвращаемого значения, имя, список параметров (возможно, пустой) и тело функции. Тело функции — это блок операторов, выполняемых при вызове функции. Переданные функции при вызове аргумента должны быть совместимы с типами соответствующих параметров.

В языке C++ функции могут быть перегружены. То есть одинаковое имя может быть использовано при определении разных функций, отличающихся количеством или типами параметров. На основании переданных при вызове аргументов компилятор автоматически выбирает наиболее подходящую версию функции. Процесс выбора правильной версии из набора перегруженных функций называют подбором функции с наилучшим соответствием.

Термины

Автоматический объект (*automatic object*). Объект, являющийся для

функции локальным. Автоматические объекты создаются и инициализируются при каждом обращении и удаляются по завершении блока, в котором они были определены.

Аргумент (argument). Значение, предоставляемое при вызове функции для инициализации соответствующих параметров.

Аргумент по умолчанию (default argument). Значение, определенное для использования, когда аргумент пропущен при вызове функции.

Бесконечная рекурсия (recursion loop). Когда у рекурсивной функции отсутствует условие остановки, она вызывает сама себя до исчерпания стека программы.

Встраиваемая функция (inline function). Функция, тело которой встраивается по месту обращения, если это возможно. Встраиваемые функции позволяют избежать обычных дополнительных затрат, поскольку их вызов заменяет код тела функции.

Вызов по значению (call by value). См. передача по значению.

Вызов по ссылке (call by reference). См. передача по ссылке.

Исполняемый файл (executable file). Файл, содержащий программный код, который может быть выполнен операционной системой.

Класс initializer_list. Библиотечный класс, представляющий разделяемый запятыми список объектов одинакового типа, заключенный в фигурные скобки.

Компоновка (link). Этап компиляции, на котором несколько объектных файлов объединяются в исполняемую программу.

Локальная переменная (local variable). Переменные, определенные в блоке.

Локальный статический объект (local static object). Локальный объект, который создается и инициализируется только один раз перед первым вызовом функции, в которой используется ее значение. Значение локального статического объекта сохраняется на протяжении всех вызовов функции.

Макрос assert. Макрос препроцессора, который получает одно выражение, используемое в качестве условия. Если переменная препроцессора NDEBUG не определена, макрос assert проверяет условие. Если оно ложно, макрос assert выводит сообщение и завершает программу.

Макрос препроцессора (preprocessor macro). Средство препроцессора, ведущее себя как встраиваемая функция. Кроме макроя assert, современные программы C++ очень редко используют макросы

препроцессора.

Наилучшее соответствие (best match). Функция, выбранная для вызова из набора перегруженных версий. Если наилучшее соответствие существует, выбранная функция лучше остальных подходит по крайней мере для одного аргумента вызова и не хуже остальных версий для оставшейся части аргументов.

Неоднозначный вызов (ambiguous call). Ошибка времени компиляции, происходящая при поиске подходящей функции, когда две или более функций обеспечивают одинаково хорошее соответствие для вызова.

Объектный код (object code). Формат, в который компилятор преобразует исходный код.

Объектный файл (object file). Файл, содержащий объектный код, созданный компилятором из предоставленного файла исходного кода. Исполняемый файл создается в результате компоновки одного или нескольких объектных файлов.

Оператор(). Оператор вызова. Запускает функцию на выполнение. Круглые скобки, следующие за именем функции или указателем на функцию, заключают разделяемый запятыми список аргументов, который может быть пуст.

Отсутствие соответствия (no match). Ошибка времени компиляции, происходящая при поиске подходящей функции, когда не обнаружено ни одной функции с параметрами, которые соответствуют аргументам при данном вызове.

Параметр (parameter). Локальная переменная, объявляемая в списке параметров функции. Параметры инициализируются аргументами, предоставляемыми при каждом вызове функции.

Перегруженная функция (overloaded function). Функция, которая имеет то же имя, что и по крайней мере одна другая функция. Перегруженные функции должны отличаться по количеству или типу их параметров.

Передача по значению (pass by value). Способ передачи аргументов параметрам не ссылочного типа. Не ссылочный параметр — это копия значения соответствующего аргумента.

Передача по ссылке (pass by reference). Способ передачи аргументов параметрам ссылочного типа. Ссылочные параметры работают так же, как и любая другая ссылка; параметр связан со своим аргументом.

Замыкающий тип возвращаемого значения (trailing return type). Тип возвращаемого значения, определенный после списка параметров.

Подбор функции (function matching). Процесс, в ходе которого

компилятор ассоциирует вызов функции с определенной версией из набора перегруженных функций. При подборе функции используемые в обращении аргументы сравниваются со списком параметров каждой версии перегруженной функции.

Подходящая функция (viable function). Подмножество перегруженных функций, которые могли бы соответствовать данному вызову. У подходящих функций количество параметров совпадает с количеством переданных при обращении аргументов, а тип каждого аргумента может быть преобразован в тип соответствующего параметра.

Поиск перегруженной функции (overload resolution). См. подбор функции.

Продолжительность существования объекта (object lifetime). Каждый объект характеризуется своей продолжительностью существования. Нестатические объекты, определенные в блоке, существуют от момента их определения и до конца блока, в котором они определены. Глобальные объекты создаются во время запуска программы. Локальные статические объекты создаются прежде, чем выполнение впервые пройдет через определение объекта. Глобальные объекты и локальные статические объекты удаляются по завершении функции `main()`.

Прототип функции (function prototype). Синоним объявления функции. В прототипе указано имя, тип возвращаемого значения и типы параметров функции. Чтобы функцию можно было вызвать, ее прототип должен быть объявлен перед точкой обращения.

Раздельная компиляция (separate compilation). Способность разделить программу на несколько отдельных файлов исходного кода.

Рекурсивная функция (recursive function). Функция, которая способна вызвать себя непосредственно или косвенно.

Скрытое имя (hidden name). Имя, объявленное в области видимости, но скрытое ранее объявлена сущностью с тем же именем, объявленным вне этой области видимости.

Тело функции (function body). Блок операторов, в котором определены действия функции.

Тип возвращаемого значения (return type). Часть объявления функции, определяющее тип значения, которое возвращает функция.

Функция `constexpr`. Функция, способная возвратить константное выражение. Функция `constexpr` неявно является встраиваемой.

Функция (function). Именованный блок действий.

Функция-кандидат (candidate function). Одна из функций набора,

рассматриваемая при поиске соответствия вызову функции. Кандидатами считаются все функции, объявленные в области видимости обращения, имя которых совпадает с используемым в обращении.

Глава 7

Классы

Классы в языке C++ используются для определения собственных типов данных. Определение типов, отражающих концепции решаемых задач, позволяет существенно упростить написание, отладку и модификацию программ.

В этой главе будет продолжено описание классов, начатое в главе 2. Основное внимание здесь уделяется важности абстракции данных, позволяющей отделять реализацию объекта от операций, в которых объект может участвовать. В главе 13 будет описано, как контролировать происходящее при копировании, перемещении, присвоении и удалении объекта, а в главе 14 рассматривается определение собственных операторов.

Фундаментальными идеями, лежащими в основе концепции *классов* (class), являются *абстракция данных* (data abstraction) и *инкапсуляция* (encapsulation). Абстракция данных — программный подход, полагающийся на разделение *интерфейса* (interface) и *реализации* (implementation). Интерфейс класса состоит из операций, которые пользователь класса может выполнить с его объектом. Реализация включает переменные-члены класса, тела функций, составляющих интерфейс, а также любые функции, которые нужны для определения класса, но не предназначены для общего использования.

Инкапсуляция обеспечивает разделение интерфейса и реализации класса. Инкапсулируемый класс скрывает свою реализацию от пользователей, которые могут использовать интерфейс, но не имеют доступа к реализации класса.

Класс, использующий абстракцию данных и инкапсуляцию, называют *абстрактным типом данных* (abstract data type). Внутренняя реализация абстрактного типа данных заботит только его разработчика. Программисты, которые используют этот класс, не обязаны ничего знать о том, как внутренне работает этот тип. Они могут рассматривать его как *абстрацию*.

7.1. Определение абстрактных типов данных

Класс `Sales_item`, использованный в главе 1, является абстрактным типом данных. При использовании объекта класса `Sales_item` задействовался его интерфейс (т.е. операции, описанные в разделе 1.5.1). Мы не имели доступа к переменным-членам, хранящимся в объекте класса `Sales_item`. На самом деле нам даже не было известно, какие переменные-члены имеет этот класс.

Наш класс `Sales_data` (см. раздел 2.6.1) не был абстрактным типом данных. Он позволяет пользователям обращаться к его переменным-членам и вынуждает пользователей писать собственные операции. Чтобы сделать класс `Sales_data` абстрактным типом, необходимо определить операции, доступные для его пользователей. Как только класс `Sales_data` определит собственные операции, мы сможем инкапсулировать (т.е. скрыть) его переменные-члены.

7.1.1. Разработка класса `Sales_data`

В конечном счете хочется, чтобы класс `Sales_data` поддержал тот же набор операций, что и класс `Sales_item`. У класса `Sales_item` была одна *функция-член* (*member function*) (см. раздел 1.5.2) по имени `isbn`, а также поддерживались операторы `+`, `=`, `+=`, `<<` и `>>`.

Определение собственных операторов рассматривается в главе 14, а пока определим обычные (именованные) функции для этих операций. По причинам, рассматриваемым в разделе 7.1.5, функции, осуществляющие сложение и операции ввода-вывода, не будут членами класса `Sales_data`. Мы определим эти функции как обычные. Функция, выполняющая составное присвоение, будет членом класса, и по причинам, рассматриваемым в разделе 7.1.5, наш класс не должен определять присвоение.

Таким образом, интерфейс класса `Sales_data` состоит из следующих операций.

- Функция-член `isbn()`, возвращающая ISBN объекта.
- Функция-член `combine()`, добавляющая один объект класса `Sales_data` к другому.
- Функция `add()`, суммирующая два объекта класса `Sales_data`.
- Функция `read()`,читывающая данные из потока `istream` в объект

класса `Sales_data`.

- Функция `print()`, выводящая значение объекта класса `Sales_data` в поток `ostream`.

Ключевая концепция. Различие в ролях программистов

Пользователями (`user`) программисты обычно называют людей, использующих их приложения. Аналогично разработчик класса реализует его для *пользователей* класса. В данном случае пользователем является другой программист, а не конечный пользователь приложения.

Когда упоминается *пользователь*, имеющееся в виду лицо определяет контекст употребления термина. Если речь идет о *пользовательском коде* или *пользователе* класса `Sales_data`, то подразумевается программист, который использует класс. Если речь идет о *пользователе* приложения книжного магазина, то подразумевается менеджер склада, использующий приложение.



Говоря о *пользователях*, программисты C++, как правило, имеют в виду как пользователей приложения, так и пользователей класса.

В простых приложениях пользователь класса вполне может быть и его разработчиком. Но даже в таких случаях имеет смысл различать роли. Разрабатывая интерфейс класса, следует думать о том, чтобы его было проще использовать. При использовании класса не нужно думать, как именно он работает.

Авторы хороших приложений добиваются успеха потому, что понимают и реализуют потребности пользователей. Точно так же хорошие разработчики класса обращают пристальное внимание на потребности программистов, которые будут использовать их класс. У хорошо разработанного класса удобный, интуитивно понятный интерфейс, а его реализация достаточно эффективна для решения задач пользователя.

Использование пересмотренного класса `Sales_data`

Прежде чем думать о реализации нашего класса, обдумаем то, как можно использовать функции его интерфейса. В качестве примера использования этих функций напишем новую версию программы

книжного магазина из раздела 1.6, работающую с объектами класса `Sales_data`, а не `Sales_item`:

```
Sales_data total;           // переменная для хранения
текущей суммы
if (read(cin, total)) {    // прочитать первую
транзакцию
    Sales_data trans;      // переменная для хранения
данных следующей
                           // транзакции
    while(read(cin, trans)) { // читать остальные
транзакции
        if (total.isbn() == trans.isbn()) // проверить
isbn
            total.combine(trans); // обновить текущую сумму
        else {
            print(cout, total) << endl; // отобразить
результаты
            total = trans; // обработать следующую книгу
        }
    }
    print(cout, total) << endl; // отобразить
последнюю транзакцию
} else {                   // ввода нет
    cerr << "No data?!" << endl; // уведомить
пользователя
}
```

Сначала определяется объект класса `Sales_data` для хранения текущей суммы. В условии оператора `if` происходит вызов функции `read()` для чтения в переменную `total` первой транзакции. Это условие работает, как и другие написанные ранее циклы с использованием оператора `>>`. Как и оператор `>>`, наша функция `read()` будет возвращать свой потоковый параметр, который и проверяет условие (см. раздел 4.11.2). Если функция `read()` потерпит неудачу, сработает часть `else`, выводящая сообщение об ошибке.

Если данные прочитаны успешно, определяем переменную `trans` для хранения всех транзакций. Условие цикла `while` также проверяет поток, возвращенный функцией `read()`. Пока операции ввода в функции `read()` успешны, условие выполняется и обрабатывается следующая

транзакция.

В цикле `while` происходит вызов функции-члена `isbn()` объектов `total` и `trans`, возвращающей их ISBN. Если объекты `total` и `trans` относятся к той же книге, происходит вызов функции `combine()`, добавляющей компоненты объекта `trans` к текущей сумме, хранящейся в объекте `total`. Если объект `trans` представляет новую книгу, происходит вызов функции `print()`, выводящей итог по предыдущей книге. Поскольку функция `print()` возвращает ссылку на свой потоковый параметр, ее результат можно использовать как левый операнд оператора `<<`. Это сделано для того, чтобы вывести символ новой строки после результата, созданного функцией `print()`. Затем объект `trans` присваивается объекту `total`, начиная таким образом обработку записи следующей книги в файле.

По исчерпании ввода следует не забыть вывести данные последней транзакции. Для этого после цикла `while` используется еще один вызов функции `print()`.

Упражнения раздела 7.1.1

Упражнение 7.1. Напишите версию программы обработки транзакций из раздела 1.6 с использованием класса `Sales_data`, созданного для упражнений в разделе 2.6.1.



7.1.2. Определение пересмотренного класса Sales_data

У пересмотренного класса будут те же переменные-члены, что и у версии, определенной в разделе 2.6.1: член типа `string` по имени `bookNo`, представляющий ISBN, член типа `unsigned` по имени `units_sold`, представляющий количество проданных экземпляров книги, и член типа `double` по имени `revenue`, представляющий общий доход от этих продаж.

Как уже упоминалось, у класса будут также две функции-члена, `combine()` и `isbn()`. Кроме того, предоставим классу `Sales_data` другую функцию-член, чтобы возвращать среднюю цену, по которой были проданы книги. Эта функция, назовем ее `avg_price()`, не предназначена для общего использования. Она будет частью реализации, а не интерфейса.

Функции-члены определяют (см. раздел 6.1) и объявляют (см. раздел 6.1.2) как обычные функции. Функции-члены *должны быть* объявлены в классе, но определены они *могут* быть непосредственно в классе или вне тела класса. Функции, не являющиеся членами класса, но являющиеся частью интерфейса, как функции `add()`, `read()` и `print()`, объявляются и определяются вне класса.

С учетом вышеизложенного напишем пересмотренную версию класса `Sales_data`:

```
struct Sales_data {
    // новые члены: операции с объектами класса
Sales_data
    std::string isbn() const { return bookNo; }
    Sales_data& combine(const Sales_data&);
    double avg_price() const;
    // те же переменные-члены, что и в р. 2.6.1
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};

// функции интерфейса класса Sales_data, не
являющиеся его членами
```

```
Sales_data add( const Sales_data&, const  
Sales_data&);  
std::ostream &print( std::ostream&, const  
Sales_data&);  
std::istream &read( std::istream&, Sales_data&);
```



Функции, определенные в классе, неявно являются встраиваемыми (см. раздел 6.5.2).

Определение функций-членов

Хотя каждый член класса должен быть объявлен в самом классе, тело функции-члена можно определить либо в, либо вне тела класса. Функция `isbn()` определяется в классе `Sales_data`, а функции `combine()` и `avg_price()` вне его.

Сначала рассмотрим функцию `isbn()`, возвращающую строку и имеющую пустой список параметров:

```
std::string isbn() const { return bookNo; }
```

Как и у любой функции, тело функции-члена является блоком. В данном случае блок содержит один оператор `return`, возвращающий значение переменной-члена `bookNo` объекта класса `Sales_data`. Интересно, как эта функция получает объект, член `bookNo` которого следует выбрать?

Указатель `this`

Давайте снова рассмотрим вызов функции-члена `isbn()`:

```
total.isbn()
```

Здесь для вызова функции-члена `isbn()` объекта `total` используется точечный оператор (см. раздел 4.6).

За одним исключением, рассматриваемым в разделе 7.6, вызов функции-члена осуществляется от имени объекта. Когда функция `isbn()` обращается к члену класса `Sales_data` (например, `bookNo`), она неявно обращается к членам того объекта, из которого была вызвана. В этом вызове функции `isbn()`, когда она возвращает значение члена `bookNo`, речь идет о члене `total.bookNo`.

Функция-член способна обратиться к тому объекту, из которого она

была вызвана, благодаря дополнительному неявному параметру `this`. Когда происходит вызов функции-члена, указатель `this` инициализируется адресом объекта, из которого была вызвана функция. Рассмотрим следующий вызов:

```
total.isbn()
```

Здесь компилятор присваивает адрес объекта `total` указателю `this` и неявно передает его как параметр функции `isbn()`. Компилятор как бы переписывает этот вызов так:

```
// псевдокод, в который преобразуется вызов
функции-члена
```

```
Sales_data::isbn(&total)
```

Этот код вызывает функцию-член `isbn()` класса `Sales_data`, передав адрес объекта `total`.

В функции-члене можно обратиться непосредственно к членам объекта, из которого она была вызвана. Для использования членов объекта, на который указывает указатель `this`, можно не использовать оператор доступа к члену. Любое непосредственное использование члена класса подразумевает использование указателя `this`. Таким образом, когда функция `isbn()` использует переменную `bookNo`, она неявно использует член объекта, на который указывает указатель `this`. Это аналогично синтаксису `this->bookNo`.

Параметр `this` определяется неявно и автоматически. Кроме того, определить параметр или переменную по имени `this` самому нельзя, но в теле функции-члена его использовать можно. Вполне допустимо, хоть и не нужно, определить функцию `isbn()` так:

```
std::string isbn() const { return this->bookNo; }
```

Поскольку указатель `this` всегда предназначен для обращения к "этому" объекту, он является константным (см. раздел 2.4.2). Нельзя изменить адрес, хранящийся в указателе `this`.

Константные функции-члены

Еще одним важным моментом функции-члена `isbn()` является ключевое слово `const`, расположенное за списком параметров. Оно применяется для модификации типа неявного указателя `this`.

По умолчанию указатель `this` имеет тип константного указателя на неконстантную версию типа класса. Например, типом по умолчанию указателя `this` в функции-члене `Sales_data` является `Sales_data *const`. Хотя указатель `this` и неявен, он подчиняется обычным

правилам инициализации, согласно которым (по умолчанию) нельзя связать указатель `this` с константным объектом (см. раздел 2.4.2). Следствием этого факта, в свою очередь, является невозможность вызвать обычную функцию-член для константного объекта.

Если бы функция `isbn()` была обычной и если бы указатель `this` был обычным параметром типа указателя, то мы объявили бы его как `const Sales_data *const`. В конце концов, тело функции `isbn()` не изменяет объект, на который указывает указатель `this`; таким образом, эта функция стала бы гибче, если бы указатель `this` был указателем на константу (см. раздел 6.2.3).

Однако указатель `this` неявный и не присутствует в списке параметров, поэтому нет места, где можно было бы указать, что он должен быть указателем на константу. Язык решает эту проблему, позволяя разместить ключевое слово `const` после списка параметров функции-члена. Это означает, что указатель `this` является указателем на константу. Функции-члены, использующие ключевое слово `const` таким образом, являются *константными функциями-членами* (*const member function*).

Тело функции `isbn()` можно считать написанным так:

```
// псевдокод, иллюстрирующий использование неявного
указателя
// этот код недопустим: нельзя явно
определить этот указатель
// обратите внимание, что это указатель на
константу, поскольку isbn()
// является константным членом класса
std::string Sales_data::isbn(const Sales_data*
*const this)
{ return this->isbn; }
```

Тот факт, что `this` является указателем на константу, означает, что константные функции-члены не могут изменить объект, для которого они вызваны. Таким образом, функция `isbn()` может читать значения переменных-членов объектов, для которых она вызывается, но не изменять их.



Константные объекты, ссылки и указатели на константные объекты могут

вызывать только константные функции-члены

Область видимости класса и функции-члены

Помните, что класс сам является областью видимости (см. раздел 2.6.1). Определения функций-членов класса находятся в области видимости самого класса. Следовательно, использованное функцией `isbn()` имя `bookNo` относится к переменной-члену, определенной в классе `Sales_data`.

Следует заметить, что функция `isbn()` может использовать имя `bookNo`, несмотря на то, что оно определено *после* функции `isbn()`. Как будет описано в разделе 7.4.1, компилятор обрабатывает классы в два этапа — сначала объявления членов класса, затем тела функций-членов, если таковые вообще имеются. Таким образом, тела функций-членов могут использовать другие члены своих классов, независимо от того, где именно в классе они определены.

Определение функции-члена вне класса

Подобно любой другой функции, при определении функции-члена вне тела класса ее определение должно соответствовать объявлению. Таким образом, тип возвращаемого значения, список параметров и имя должны совпадать с объявлением в теле класса. Если член класса был объявлен как константная функция, то в определении после списка параметров также должно присутствовать ключевое слово `const`. Имя функции-члена, определенное вне класса, должно включить имя класса, которому она принадлежит:

```
double Sales_data::avg_price() const {
    if (units_sold)
        return revenue / units_sold;
    else
        return 0;
}
```

Имя функции, `Sales_data::avg_price()`, использует оператор области видимости (см. раздел 1.2), чтобы указать, что определяемая функция по имени `avg_price` объявлена в пределах класса `Sales_data`. Как только компилятор увидит имя функции, остальная часть кода интерпретируется как относящаяся к области видимости класса. Таким образом, когда функция `avg_price()` обращается к переменным `revenue` и `units_sold`, она неявно имеет в виду члены класса

`Sales_data.`

Определение функции, возвращающей указатель `this` на объект

Функция `combine()` должна действовать как составной оператор присвоения `+=`. Объект, для которого вызвана эта функция, представляет собой левый операнд присвоения. Правый операнд передается как аргумент явно:

```
Sales_data& Sales_data::combine(const Sales_data
&rhs) {
    units_sold += rhs.units_sold; // добавить члены
объекта rhs
    revenue += rhs.revenue;           // к членам объекта
'this'
    return *this; // вернуть объект, для которого
вызвана функция
}
```

Когда наша программа обработки транзакций осуществляет вызов `total.combine(trans); // обновить текущую сумму` адрес объекта `total` находится в неявном параметре `this`, а объект `trans` связан с параметром `rhs`. Таким образом, при вызове функции `combine()` выполняется следующий оператор:

```
units_sold += rhs.units_sold; // добавить члены
объекта rhs
```

В результате произойдет сложение переменных `total.units_sold` и `trans.units_sold`, а сумма должна сохраниться снова в переменной `total.units_sold`.

Самым интересным в этой функции является тип возвращаемого значения и оператор `return`. Обычно при определении функции, работающей как стандартный оператор, она должна подражать поведению этого оператора. Стандартные операторы присвоения возвращают свой левый operand как l-значение (см. раздел 144). Чтобы вернуть l-значение, наша функция `combine()` должна вернуть ссылку (см. раздел 6.3.2). Поскольку левый operand — объект класса `Sales_data`, тип возвращаемого значения — `Sales_data&`.

Как уже упоминалось, для доступа к члену объекта, функция-член которого выполняется, необязательно использовать неявный указатель `this`. Однако для доступа к объекту в целом указатель `this`

действительно нужен:

```
return *this; // возвратить объект, для которого  
вызвана функция
```

Здесь оператор `return` обращается к значению указателя `this`, чтобы получить объект, функция которого выполняется. Таким образом, для этого вызова возвращается ссылка на объект `total`.

Упражнения раздела 7.1.2

Упражнение 7.2. Добавьте функции-члены `combine()` и `isbn()` в класс `Sales_data`, который был написан для упражнений из раздела 2.6.2.

Упражнение 7.3. Пересмотрите свою программу обработки транзакций из раздела 7.1.1 так, чтобы использовать эти функции-члены.

Упражнение 7.4. Напишите класс по имени `Person`, представляющий имя и адрес человека. Используйте для содержания каждого из этих членов тип `string`. В последующих упражнениях в этот класс будут добавлены новые средства.

Упражнение 7.5. Снабдите класс `Person` операциями возвращения имени и адреса. Должны ли эти функции быть константами? Объясните свой выбор.



7.1.3. Определение функций, не являющихся членом класса, но связанных с ним

Авторы классов нередко определяют такие вспомогательные функции, как наши функции `add()`, `read()` и `print()`. Хотя определяемые ими операции концептуально являются частью интерфейса класса, частью самого класса они не являются.

Мы определяем функции, не являющиеся членом класса, как любую другую функцию, т.е. ее объявление обычно отделено от определения (см. раздел 6.1.2). Функции, концептуально являющиеся частью класса, но не определенные в нем, как правило, объявляются (но не определяются) в том же заголовке, что и сам класс. Таким образом, чтобы использовать любую часть интерфейса, пользователю достаточно включить только один файл.



Обычно функция, не являющаяся членом класса, но из состава его интерфейса объявляется в том же заголовке, что и сам класс.

Определение функций `read()` и `print()`

Функции `read()` и `print()` выполняют ту же задачу, что и код в разделе 2.6.2, поэтому и не удивительно, что тела этих функций очень похожи на код, представленный там:

```
// введенные транзакции содержат ISBN, количество
проданных книг и
// цену книги
istream &read(istream &is, Sales_data &item) {
    double price = 0;
    is >> item.bookNo >> item.units_sold >> price;
    item.revenue = price * item.units_sold;
    return is;
}

ostream &print(ostream &os, const Sales_data &item)
{
    os << item.isbn() << " " << item.units_sold << " "
        << item.revenue << " " << item.avg_price();
    return os;
}
```

Функция `read()` читает данные из предоставленного потока в заданный объект. Функция `print()` выводит содержимое предоставленного объекта в заданный поток.

В этих функциях, однако, следует обратить внимание на два момента. Во-первых, обе функции получают ссылки на соответствующие объекты классов ввода и вывода. Классы ввода и вывода — это типы, не допускающие копирования, поэтому их передача возможна только по ссылке (см. раздел 6.2.2). Кроме того, чтение и запись в поток изменяют его, поэтому обе функции получают обычные ссылки, а не ссылки на константы.

Второй заслуживающий внимания момент: функция `print()` не

выводит новую строку. Обычно функции вывода осуществляют минимальное форматирование. Таким образом, пользовательский код может сам решить, нужна ли новая строка.

Определение функции add()

Функция `add()` получает два объекта класса `Sales_data` и возвращает новый объект класса `Sales_data`, представляющий их сумму:

```
Sales_data add( const Sales_data &lhs, const
Sales_data &rhs) {
    Sales_data sum = lhs; // копирование переменных-
    // членов из lhs в sum
    sum.combine(rhs); // добавить переменные-члены
    rhs в sum
    return sum;
}
```

В теле функции определяется новый объект класса `Sales_data` по имени `sum`, предназначенный для хранения суммы двух транзакций. Инициализируем объект `sum` копией объекта `lhs`. По умолчанию копирование объекта класса подразумевает копирование и членов этого объекта. После копирования у членов `bookNo`, `units_sold` и `revenue` объекта `sum` будут те же значения, что и у таковых объекта `lhs`. Затем происходит вызов функции `combine()`, суммирующей значения переменных-членов `units_sold` и `revenue` объектов `rhs` и `sum` в последний. По завершении возвращается копия объекта `sum`.

Упражнения раздела 7.1.3

Упражнение 7.6. Определите собственные версии функций `add()`, `read()` и `print()`.

Упражнение 7.7. Перепишите программу обработки транзакций, написанной для упражнений в разделе 7.1.2, так, чтобы использовать эти новые функции.

Упражнение 7.8. Почему функция `read()` определяет свой параметр типа `Sales_data` как простую ссылку, а функция `print()` — как ссылку на константу?

Упражнение 7.9. Добавьте в код, написанный для упражнений в разделе 7.1.2, операции чтения и вывода объектов класса `Person`.

Упражнение 7.10. Что делает условие в следующем операторе `if`?

```
if ( read( read( cin, data1) , data2) )
```



7.1.4. Конструкторы

Каждый класс определяет, как могут быть инициализированы объекты его типа. Класс контролирует инициализацию объекта за счет определения одной или нескольких специальных функций-членов, известных как конструкторы (*constructor*). Задача конструктора — инициализировать переменные-члены объекта класса. Конструктор выполняется каждый раз, когда создается объект класса.

В этом разделе рассматриваются основы определения конструкторов. Конструкторы — удивительно сложная тема. На самом деле мы сможем больше сказать о конструкторах в разделах 7.5, 15.7 и 18.1.3, а также в главе 13.

Имя конструкторов совпадает с именем класса. В отличие от других функций, у конструкторов нет типа возвращаемого значения. Как и другие функции, конструкторы имеют (возможно пустой) список параметров и (возможно пустое) тело. У класса может быть несколько конструкторов. Подобно любой другой перегруженной функции (см. раздел 6.4), конструкторы должны отличаться друг от друга количеством или типами своих параметров.

В отличие от других функций-членов, конструкторы не могут быть объявлены константами (см. раздел 7.1.2). При создании константного объекта типа класса его константность не проявится, пока конструктор не закончит инициализацию объекта. Таким образом, конструкторы способны осуществлять запись в константный объект во время его создания.



Синтезируемый стандартный конструктор

Хотя в нашем классе `Sales_data` не определено конструкторов, использующие его программы компилировались и выполнялись правильно. Например, программа из раздела 7.1.1 определяла два объекта класса `Sales_data`:

```
Sales_data total; // переменная для хранения
текущей суммы
Sales_data trans; // переменная для хранения данных
```

следующей

// транзакции

Естественно, возникает вопрос: как инициализируются объекты `total` и `trans`?

Настолько известно, инициализатор для этих объектов не предоставлялся, поэтому они инициализируются значением по умолчанию (см. раздел 2.2.1). Классы сами контролируют инициализацию по умолчанию, определяя специальный конструктор, известный как *стандартный конструктор* (default constructor). Стандартным считается конструктор, не получающий никаких аргументов.

Как будет продемонстрировано, стандартный конструктор является особенным во многом, например, если класс не определяет конструкторы явно, компилятор сам определит стандартный конструктор *неявно*.

Созданный компилятором конструктор известен как *синтезируемый стандартный конструктор* (synthesized default constructor). У большинства классов этот синтезируемый конструктор инициализирует каждую переменную-член класса следующим образом:

- Если есть внутриклассовый инициализатор (см. раздел 2.6.1), он и используется для инициализации члена класса.
- В противном случае член класса инициализируется значением по умолчанию (см. раздел 2.2.1).

Поскольку класс `Sales_data` предоставляет инициализаторы для переменных `units_sold` и `revenue`, синтезируемый стандартный конструктор использует данные значения для инициализации этих членов. Переменная `bookNo` инициализируется значением по умолчанию, т.е. пустой строкой.

Некоторые классы не могут полагаться на синтезируемый стандартный конструктор

Только довольно простые классы, такие как текущий класс `Sales_data`, могут полагаться на синтезируемый стандартный конструктор. Как правило, собственный стандартный конструктор для класса определяют потому, что компилятор создает его, *только если для класса не определено никаких других конструкторов*. Если определен хоть один конструктор, то у класса не будет стандартного конструктора, если не определить его самостоятельно. Основание для этого правила таково: если класс требует контроля инициализации объекта в одном случае, то он, вероятно, потребует его во всех случаях.



Компилятор создает стандартный конструктор автоматически, только если в классе *не объявлено* никаких конструкторов.

Вторая причина для определения стандартного конструктора в том, что у некоторых классов синтезируемый стандартный конструктор работает неправильно. Помните, что определенные в блоке объекты встроенного или составного типа (такого как массивы и указатели) без инициализации имеют неопределенное значение (см. раздел 2.2.1). Это же относится к не инициализированным членам встроенного типа. Поэтому классы, у которых есть члены встроенного или составного типа, должны либо инициализировать их в классе, либо определять собственную версию стандартного конструктора. В противном случае пользователи могли бы создать объекты с членами, значения которых не определены.



Классы, члены которых имеют встроенный или составной тип, могут полагаться на синтезируемый стандартный конструктор, *только если* у всех таких членов есть внутриклассовые инициализаторы.

Третья причина определения некоторыми классами собственного стандартного конструктора в том, что иногда компилятор неспособен создать его. Например, если у класса есть член типа класса и у этого класса нет стандартного конструктора, то компилятор не сможет инициализировать этот член. Для таких классов следует определить собственную версию стандартного конструктора. В противном случае у класса не будет пригодного для использования стандартного конструктора. Дополнительные обстоятельства, препятствующие компилятору создать соответствующий стандартный конструктор, приведены в разделе 13.1.6.

Определение конструкторов класса Sales_data

Определим для нашего класса `Sales_data` четыре конструктора со следующими параметрами:

- Типа `istream&`, для чтения транзакции.

- Типа `const string&` для ISBN; типа `unsigned` для количества проданных книг; типа `double` для цены проданной книги.
- Типа `const string&` для ISBN. Для других членов этот конструктор будет использовать значения по умолчанию.
- Без параметров (т.е. стандартный конструктор). Этот конструктор придется определить, поскольку определены другие конструкторы.

Добавим эти члены в класс так:

```
struct Sales_data {
    // добавленные конструкторы
    Sales_data() = default;
    Sales_data(const std::string &s): bookNo(s) { }
    Sales_data(const std::string &s, unsigned n,
double p):
        bookNo(s), units_sold(n), revenue(p*n)
    { }
    Sales_data(std::istream &); // другие члены, как прежде
    std::string isbn() const { return bookNo; }
    Sales_data& combine(const Sales_data&);
    double avg_price() const;
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

Что значит = default

Начнем с объяснения стандартного конструктора:

```
Sales_data() = default;
```

В первую очередь обратите внимание на то, что это определение стандартного конструктора, поскольку он не получает никаких аргументов. Мы определяем этот конструктор *только потому*, что хотим предоставить другие конструкторы, но и стандартный конструктор тоже нужен. Этот конструктор должен делать то же, что и синтезируемая версия.



По новому стандарту, если необходимо стандартное поведение, можно попросить компилятор создать конструктор автоматически, указав после списка параметров часть `= default`. Синтаксис `= default` может присутствовать как в объявлении в теле класса, так и в определении вне

его. Подобно любой другой функции, если часть = default присутствует в теле класса, стандартный конструктор будет встраиваемым; если она присутствует в определении вне класса, то по умолчанию этот член не будет встраиваемым.



Стандартный конструктор работает в классе Sales_data только потому, что предоставлены инициализаторы для переменных-членов встроенного типа. Если ваш компилятор не поддерживает внутриклассовые инициализаторы, для инициализации каждого члена класса стандартный конструктор должен использовать список инициализации конструктора (описанный непосредственно ниже).

Список инициализации конструктора

Теперь рассмотрим два других конструктора, которые были определены в классе:

```
Sales_data( const std::string &s ) : bookNo( s ) { }
Sales_data( const std::string &s, unsigned n, double
p ) :
    bookNo( s ), units_sold( n ), revenue( p*n ) { }
```

Новой частью этих определений являются двоеточие и код между ним и фигурными скобками, обозначающими пустые тела функции. Эта новая часть — *список инициализации конструктора* (constructor initializer list), определяющий исходные значения для одной или нескольких переменных-членов создаваемого объекта. Инициализатор конструктора — это список имен переменных-членов класса, каждое из которых сопровождается исходным значением в круглых (или фигурных) скобках. Если инициализаций несколько, они отделяются запятыми.

Конструктор с тремя параметрами использует первые два параметра для инициализации переменных-членов bookNo и units_sold. Инициализатор для переменной revenue вычисляется при умножении количества проданных книг на их цену.

Конструктор с одним параметром типа string использует ее для инициализации переменной-члена bookNo, но переменные units_sold и revenue не инициализируются явно. Когда член класса отсутствует в

списке инициализации конструктора, он инициализируется неявно, с использованием того же процесса, что и у синтезируемого стандартного конструктора. В данном случае эти члены инициализируются внутриклассовыми инициализаторами. Таким образом, получающий строку конструктор эквивалентен следующему.

// то же поведение, что и у исходного конструктора
выше

```
Sales_data( const std::string &s):  
    bookNo(s), units_sold(0), revenue(0) {}
```

Обычно для конструктора лучше использовать внутриклассовый инициализатор, если он есть и присваивает члену класса правильное значение. С другой стороны, если ваш компилятор еще не поддерживает внутриклассовые инициализаторы, то каждый конструктор должен явно инициализировать каждый член встроенного типа.

Рекомендуем

Конструкторы не должны переопределять внутриклассовые инициализаторы, кроме как при использовании иного исходного значения. Если вы не можете использовать внутриклассовые инициализаторы, каждый конструктор должен явно инициализировать каждый член встроенного типа.

Следует заметить, что у обоих этих конструкторов тела пусты. Единственное, что должны сделать эти конструкторы, — присвоить значения переменным-членам. Если ничего другого делать не нужно, то тело функции пусто.

Определение конструктора вне тела класса

В отличие от наших других конструкторов, конструктору, получающему поток `istream`, действительно есть что делать. В своем теле этот конструктор вызывает функцию `read()`, чтобы присвоить переменным-членам новые значения:

```
Sales_data::Sales_data(std::istream &is) {  
    read(is, *this); // read читает транзакцию из is в  
    текущий объект  
}
```

У конструкторов нет типа возвращаемого значения, поэтому определение начинается с имени функции. Подобно любой другой

функции-члену, при определении конструктора за пределами тела класса необходимо указать класс, которому принадлежит конструктор. Таким образом, синтаксис `Sales data::Sales_data` указывает, что мы определяем член класса `Sales_data` по имени `Sales_data`. Этот член класса является конструктором, поскольку его имя совпадает с именем класса.

В этом конструкторе нет списка инициализации конструктора, хотя с технической точки зрения было бы правильней сказать, что список инициализации конструктора пуст. Даже при том, что список инициализации конструктора пуст, члены этого объекта инициализируются прежде, чем выполняется тело конструктора.

Члены, отсутствующие в списке инициализации конструктора, инициализируются соответствующим внутриклассовым инициализатором (если он есть) или значением по умолчанию. Для класса `Sales_data` это означает, что при запуске тела функции на выполнение переменная `bookNo` будет содержать пустую строку, а переменные `units_sold` и `revenue` — значение 0.

Чтобы стало понятней, напомним, что второй параметр функции `read()` является ссылкой на объект класса `Sales_data`. В разделе 7.1.2 мы обращали внимание на то, что указатель `this` используется для доступа к объекту в целом, а не к егоциальному члену. В данном случае для передачи "этого" объекта в качестве аргумента функции `read()` используется синтаксис `*this`.

Упражнения раздела 7.1.4

Упражнение 7.11. Добавьте в класс `Sales_data` конструкторы и напишите программу, использующую каждый из них.

Упражнение 7.12. Переместите определение конструктора `Sales_data()`, получающего объект `istream`, в тело класса `Sales_data`.

Упражнение 7.13. Перепишите программу из раздела 7.1.1 так, чтобы использовать конструктор с параметром `istream`.

Упражнение 7.14. Напишите версию стандартного конструктора, явно инициализирующую переменные-члены значениями, предоставленными внутриклассовыми инициализаторами.

Упражнение 7.15. Добавьте соответствующие конструкторы в класс `Person`.



7.1.5. Копирование, присвоение и удаление

Кроме определения способа инициализации своих объектов, классы контролируют также то, что происходит при копировании, присвоении и удалении объектов класса. Объекты копируются во многих случаях: при инициализации переменной, при передаче или возвращении объекта по значению (см. раздел 6.2.1 и раздел 6.3.2). Объекты присваиваются при использовании оператора присвоения (см. раздел 4.4). Объекты удаляются, когда они прекращают существование, например, при выходе локального объекта из блока, в котором он был создан (см. раздел 6.1.1). Объекты, хранимые в векторе (или массиве), удаляются при удалении вектора (или массива).

Если мы не определим эти операции, компилятор создаст их сам. Обычно создаваемые компилятором версии выполняются, копируя, присваивая или удаляя каждую переменную-член объекта. Например, когда в приложении книжного магазина (см. раздел 7.1.1) компилятор выполняет следующее присвоение:

```
total = trans; // обработать следующую книгу  
оно выполняется, как будто было написано так:  
// присвоение по умолчанию для Sales_data  
эквивалентно следующему:  
total.bookNo = trans.bookNo;  
total.units_sold = trans.units_sold;  
total.revenue = trans.revenue;
```

Более подробная информация об определении собственных версий этих операторов приведена в главе 13.



Некоторые классы не могут полагаться на синтезируемые версии

Хотя компилятор и создает сам операторы копирования, присвоения и удаления, важно понимать, что у некоторых классов их стандартные версии ведут себя неправильно. В частности, синтезируемые версии вряд ли будут правильно работать с классами, которые резервируют ресурсы, располагающиеся вне самих объектов класса. Пример резервирования и управления динамической памятью приведен в главе 12. Как будет

продемонстрировано в разделе 13.6, классы, которые управляют динамической памятью, вообще не могут полагаться на синтезируемые версии этих операций.

Однако следует заметить, что большинство классов, нуждающихся в динамической памяти, способны (и должны) использовать классы `vector` или `string`, если им нужно управляемое хранение. Классы, использующие векторы и строки, избегают сложностей, связанных с резервированием и освобождением памяти.

Кроме того, синтезируемые версии операторов копирования, присвоения и удаления правильно работают для классов, у которых есть переменные-члены класса `vector` или `string`. При копировании или присвоении объекта, обладающего переменной-членом класса `vector`, этот класс сам позаботится о копировании и присвоении своих элементов. Когда объект удаляется, переменная-член класса `vector` тоже удаляется, что в свою очередь удаляет элементы вектора. Класс `string` работает аналогично.



Пока вы еще не знаете, как определить операторы, описанные в главе 13, ресурсы, резервируемые вашими классами, должны храниться непосредственно как переменные-члены класса.



7.2. Управление доступом и инкапсуляция

На настоящий момент для нашего класса определен интерфейс; однако ничто не вынуждает пользователей использовать его. Наш класс еще не использует инкапсуляцию — пользователи вполне могут обратиться к объекту `Sales_data` и воспользоваться его реализацией. Для обеспечения инкапсуляции в языке C++ используют *спецификаторы доступа* (access specifier).

- Члены класса, определенные после спецификатора `public`, доступны для всех частей программы. *Открытые члены* (public member) определяют интерфейс к классу.

- Члены, определенные после спецификатора `private`, являются *закрытыми* (private member), они доступны для функций-членов класса, но

не доступны для кода, который использует класс. Разделы `private` инкапсулируют (т.е. скрывают) реализацию.

Переопределив класс `Sales_data` еще раз, получаем следующее:

```
class Sales_data {
public: // добавлен спецификатор доступа
    Sales_data() = default;
    Sales_data( const std::string &s, unsigned n,
double p):
        bookNo(s), units_sold(n), revenue(p*n) { }
    Sales_data( const std::string &s): bookNo(s) { }
    Sales_data( std::istream& );
    std::string isbn() const { return bookNo; }
    Sales_data &combine( const Sales_data& );
private: // добавлен спецификатор доступа
    double avg_price() const
    { return units_sold ? revenue/units_sold : 0; }
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

Конструкторы и функции-члены, являющиеся частью интерфейса (например, `isbn()` и `combine()`), должны располагаться за спецификатором `public`; переменные-члены и функции, являющиеся частью реализации, располагаются за спецификатором `private`.

Класс может содержать любое количество спецификаторов доступа; нет никаких ограничений на то, как часто используется спецификатор. Каждый спецификатор определяет уровень доступа последующих членов. Заданный уровень доступа остается в силе до следующего спецификатора доступа или до конца тела класса.

Использование ключевых слов `class` и `struct`

Было также внесено еще одно изменение: в начале определения класса использовано ключевое слово `class`, а не `struct`. Это изменение является чисто стилистическим; тип класса можно определить при помощи любого из этих ключевых слов. Единственное различие между ключевыми словами `struct` и `class` в заданном по умолчанию уровне доступа.

Члены класса могут быть определены перед первым спецификатором доступа. Уровень доступа к таким членам будет зависеть от того, как

определяется класс. Если используется ключевое слово `struct`, то члены, определенные до первого спецификатора доступа, будут открытыми; если используется ключевое слово `class`, то они будут закрытыми.

Общепринятым стилем считается определение классов, все члены которого предположительно будут открытыми, с использованием ключевого слова `struct`. Если члены класса должны быть закрытыми, используется ключевое слово `class`.



Единственное различие между ключевыми словами `class` и `struct` в задаваемом по умолчанию уровне доступа.

Ключевая концепция. Преимущества инкапсуляции

Инкапсуляция предоставляет два важных преимущества.

- Пользовательский код не может по неосторожности повредить состояние инкапсулированного объекта.
- Реализация инкапсулированного класса может со временем измениться, это не потребует изменений в коде на пользовательском уровне.

Определив переменные-члены закрытыми, автор класса получает возможность вносить изменения в данные. Если реализация изменится, то вызванные этим последствия можно исследовать только в коде класса. Пользовательский код придется изменять только при изменении интерфейса. Если данные являются открытыми, то любой использовавший их код может быть нарушен. Пришлось бы найти и переписать любой код, который полагался на прежнюю реализацию, и только затем использовать программу.

Еще одно преимущество объявления переменных-членов закрытыми в том, что данные защищены от ошибок, которые могли бы внести пользователи. Если есть ошибка, повреждающая состояние объекта, места ее поиска ограничены только тем кодом, который является частью реализации. Это существенно облегчает поиск проблем и обслуживание программы.

Упражнения раздела 7.2

Упражнение 7.16. Каковы ограничения (если они есть) на количество

спецификаторов доступа в определении класса? Какие виды членов должны быть определены после спецификатора `public`? Какие после спецификатора `private`?

Упражнение 7.17. Каковы различия (если они есть) между ключевыми словами `class` и `struct`?

Упражнение 7.18. Что такое инкапсуляция? Чем она полезна?

Упражнение 7.19. Укажите, какие члены класса `Person` имеет смысл объявить как `public`, а какие как `private`. Объясните свой выбор.



7.2.1. Друзья

Теперь, когда переменные-члены класса `Sales_data` стали закрытыми, функции `read()`, `print()` и `add()` перестали компилироваться. Проблема в том, что хоть эти функции и являются частью интерфейса класса `Sales_data`, его членами они не являются.

Класс может позволить другому классу или функции получить доступ к своим не открытым членам, установив для них *дружественные отношения* (`friend`). Класс объявляет функцию дружественной, включив ее объявление с предваряющим ключевым словом `friend`:

```
class Sales_data {
    // добавлены объявления дружественных функций, не
являющихся
    // членами класса Sales_data
    friend Sales_data add( const Sales_data&, const
Sales_data& );
    friend std::istream &read( std::istream&, Sales_data& );
    friend std::ostream &print( std::ostream&, const
Sales_data& );
    // другие члены и спецификаторы доступа, как
прежде
public:
    Sales_data() = default;
    Sales_data( const std::string &s, unsigned n,
double p ):
        bookNo( s ), units_sold( n ), revenue( p*n )
```

```

{ }

Sales_data( const std::string &s): bookNo(s) { }
Sales_data( std::istream& );
std::string isbn() const { return bookNo; }
Sales_data &combine( const Sales_data&);

private:
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};

// объявления частей, не являющихся членами
interface
// класса Sales_data
Sales_data add( const Sales_data&, const
Sales_data& );
std::istream &read( std::istream&, Sales_data& );
std::ostream &print( std::ostream&, const
Sales_data& );

```

Объявления друзей могут располагаться только в определении класса; использоваться они могут в классе повсюду. Друзья не являются членами класса и не подчиняются спецификаторам доступа раздела, в котором они объявлены. Более подробная информация о дружественных отношениях приведена в разделе 7.3.4.



Объявления друзей имеет смысл группировать в начале или в конце определения класса.



Хотя пользовательский код не должен изменяться при изменении определения класса, файлы исходного кода, использующие этот класс, следует перекомпилировать при каждом изменении класса.



Объявление дружественных отношений

Объявление дружественных отношений устанавливает только право доступа. Это не объявление функции. Если необходимо, чтобы пользователи класса были в состоянии вызвать дружественную функцию, ее следует также объявить.

Чтобы сделать друзей класса видимыми его пользователям, их обычно объявляют вне класса в том же заголовке, что и сам класс. Таким образом, в заголовке `Sales_data` следует предоставить отдельные объявления (кроме объявлений дружественными в теле класса) для функций `read()`, `print()` и `add()`.



Многие компиляторы не выполняют правило, согласно которому дружественные функции должны быть объявлены *вне* класса, прежде чем они будут применены.

Некоторые компиляторы позволяют вызывать дружественную функцию, когда для нее нет обычного объявления. Даже если ваш компилятор позволяет такие вызовы, имеет смысл предоставлять отдельные объявления для дружественных функций. Так не придется переделывать весь код, если вы перейдете на компилятор, который выполняет это правило.

Упражнения раздела 7.2.1

Упражнение 7.20. Когда полезны дружественные отношения? Укажите преимущества и недостатки их использования.

Упражнение 7.21. Измените свой класс `Sales_data` так, чтобы скрыть его реализацию. Написанные вами программы, которые использовали операции класса `Sales_data`, должны продолжить работать. Перекомпилируйте эти программы с новым определением класса, чтобы проверить, остались ли они работоспособными.

Упражнение 7.22. Измените свой класс `Person` так, чтобы скрыть его реализацию.

7.3. Дополнительные средства класса

Хотя класс `Sales_data` довольно прост, он все же позволил исследовать немало средств поддержки классов. В этом разделе рассматриваются некоторые из дополнительных средств, связанных с классом, которые класс `Sales_data` не будет использовать. К этим средствам относятся *типы-члены* (*type member*), внутриклассовые инициализаторы для типов-членов класса, изменяемые переменные-члены, встраиваемые функции-члены, функции-члены, возвращающие `*this`, а также подробности определения и использования типов класса и дружественных классов.

7.3.1. Снова о членах класса

Для исследования некоторых из дополнительных средств определим пару взаимодействующих классов по имени `Screen` и `Window_mgr`.

Определение типов-членов

Класс `Screen` представляет окно на экране. У каждого объекта класса `Screen` есть переменная-член типа `string`, хранящая содержимое окна и три переменные-члена типа `string::size_type`, представляющие позицию курсора, высоту и ширину окна.

Кроме переменных и функций-членов, класс может определять собственные локальные имена таких типов. Определенные классом имена типов подчиняются тем же правилам доступа, что и любой другой его член, и могут быть открытыми или закрытыми:

```
class Screen {  
public:  
    typedef std::string::size_type pos;  
private:  
    pos cursor = 0;  
    pos height = 0, width = 0;  
    std::string contents;  
};
```

Тип `pos` определен в части `public` класса `Screen`, поскольку пользователи должны использовать это имя. Пользователи класса `Screen` не обязаны знать, что он использует класс `string` для хранения своих данных. Определив тип `pos` как открытый член, эту подробность реализации класса `Screen` можно скрыть.

В объявлении типа `pos` есть два интересных момента. Во-первых, хоть здесь и был использован оператор `typedef` (см. раздел 2.5.1), с таким же успехом можно использовать псевдоним типа (см. раздел 2.5.1):

```
class Screen {  
public:  
    // альтернативный способ объявления типа-члена с  
    // использованием  
    // псевдонима типа  
    using pos = std::string::size_type;  
    // другие члены как прежде  
};
```

Во-вторых, по причинам, которые будут описаны в разделе 7.3.4, в отличие от обычных членов, типы-члены определяются прежде, чем используются. В результате типы-члены обычно располагают в начале класса.

Функции-члены класса Screen

Чтобы сделать наш класс полезней, добавим в него конструктор, позволяющий пользователям задавать размер и содержимое экрана, наряду с членами, позволяющими переместить курсор и получить символ в указанной позиции:

```
class Screen {  
public:  
    typedef std::string::size_type pos;  
    Screen() = default; // необходим, поскольку у  
класса Screen есть  
                           // другой конструктор  
    // внутриклассовый инициализатор инициализирует  
курсор значением 0  
    Screen( pos ht, pos wd, char c) : height(ht),  
width(wd),  
                           contents(ht * wd,  
c) {}  
    char get() const // получить символ в курсоре  
    { return contents[cursor]; } // неявно  
встраиваемая  
    inline char get( pos ht, pos wd) const; // явно  
встраиваемая  
    Screen &move( pos r, pos c); // может быть сделана  
встраиваемой позже  
private:  
    pos cursor = 0;  
    pos height = 0, width = 0;  
    std::string contents;  
};
```

Поскольку мы предоставляем конструктор, компилятор не будет автоматически создавать стандартный конструктор сам. Если у нашего класса должен быть стандартный конструктор, то придется создать его явно. В данном случае используется синтаксис = default, чтобы попросить компилятор самому создать определение стандартного

конструктора (см. раздел 7.1.4).

Стоит также обратить внимание на то, что второй конструктор (получающий три аргумента) неявно использует внутриклассовый инициализатор для переменной-члена `cursor` (см. раздел 7.1.4). Если бы у класса не было внутриклассового инициализатора для переменной-члена `cursor`, то мы явно инициализировали бы ее наряду с другими переменными-членами.

Встраиваемые члены класса

У классов зачастую бывают небольшие функции, которые выгодно сделать встраиваемыми. Как уже упоминалось, определенные в классе функции-члены автоматически являются встраиваемыми (`inline`) (см. раздел 6.5.2). Таким образом, конструкторы класса `Screen` и версия функции `get()`, возвращающей обозначенный курсором символ, являются встраиваемыми по умолчанию.

Функцию-член можно объявить встраиваемой явно в ее объявлении в теле класса. В качестве альтернативы функцию можно указать встраиваемой в определении, расположенном вне тела класса:

```
inline // функцию можно указать встраиваемой в
определении
```

```
Screen &Screen::move( pos r, pos c) {
    pos row = r * width; // вычислить положение ряда
    cursor = row + c;      // переместить курсор к
    столбцу этого ряда
    return *this;           // возвратить этот объект как
    1-значение
}
char Screen::get( pos r, pos c) const // объявить
встраиваемый в классе
{
    pos row = r * width;           // вычислить положение
    ряда
    return contents[ row + c]; // возвратить символ в
    данном столбце
}
```

Хоть и не обязательно делать это, вполне допустимо указать ключевое слово `inline` и в объявлении, и в определении. Однако указание ключевого слова `inline` в определении только вне класса может

облегчить чтение класса.



По тем же причинам, по которым встраиваемые функции определяют в заголовках (см. раздел 6.5.2), встраиваемые функции-члены следует определить в том же заголовке, что и определение соответствующего класса.

Перегрузка функций-членов

Подобно функциям, которые не являются членами класса, функции-члены могут быть перегружены (см. раздел 6.4), если они отличаются количеством и/или типами параметров. При вызове функции-члена используется тот же процесс подбора функции (см. раздел 6.4), что и у функций, не являющихся членом класса.

Например, в классе Screen определены две версии функции `get()`. Одна версия возвращает символ, обозначенный в настоящее время курсором; другая возвращает символ в указанной позиции, определенной ее рядом и столбцом. Чтобы определить применяемую версию, компилятор использует количество аргументов:

```
Screen myscreen;
char ch = myscreen.get(); // вызов Screen::get()
ch = myscreen.get(0, 0); // вызов Screen::get(pos,
```

Изменяемые переменные-члены

Иногда (но не очень часто) у класса есть переменная-член, которую следует сделать изменяемой даже в константной функции-члене. Для обозначения таких членов в их объявление включают ключевое слово `mutable`.

Изменяемая переменная-член (`mutable data member`) никогда не бывает константой, даже когда это член константного объекта. Соответственно константная функция-член может изменить изменяемую переменную-член. В качестве примера добавим в класс Screen изменяемую переменную-член `access_ctr`, используемую для отслеживания частоты вызова каждой функции-члена класса Screen:

```
class Screen {
```

```

public:
    void some_member() const;
private:
    mutable size_t access_ctr; // может измениться
даже в константном
                                                // объекте
    // другие члены как прежде
};

void Screen::some_member() const {
    ++access_ctr; // сохранить количество вызовов
любой функции-члена
                                                // безотносительно других
выполняемых ею действий
}

```

Несмотря на то что функция-член `some_member()` константная, она может изменить значение переменной-члена `access_ctr`. Этот член класса является изменяемым, поэтому любая функция-член, включая константные, может изменить это значение.

Инициализаторы переменных-членов класса



Кроме класса `Screen`, определим также класс диспетчера окон, который представляет коллекцию окон на данном экране. У этого класса будет вектор объектов класса `Screen`, каждый элемент которого представляет отдельное окно. По умолчанию класс `Window_mgr` должен изначально содержать один объект класса `Screen`, инициализированный значением по умолчанию. По новому стандарту наилучшим способом определения такого значения по умолчанию является внутриклассовый инициализатор (см. раздел 2.6.1):

```

class Window_mgr {
private:
    // по умолчанию отслеживающий окна объект класса
Window_mgr
    // содержит одно пустое окно стандартного размера
    std::vector<Screen> screens{ Screen( 24, 80, ' ' ) };
};

```

При инициализации переменных-членов типа класса их конструктору

следует предоставить аргументы. В этом случае применяется список инициализации переменной-члена типа `vector` (см. раздел 3.3.1) с инициализатором для одного элемента. Этот инициализатор содержит значение типа `Screen`, передаваемое конструктору `vector<Screen>` для создания вектора с одним элементом. Это значение создается конструктором класса `Screen`, получающим параметры в виде двух размерностей и заполняющего символа, чтобы создать пустое окно заданного размера.

Как уже упоминалось, для внутриклассовой инициализации может использоваться форма инициализации `=` (как при инициализации переменных-членов класса `Screen`) или прямая форма инициализации с использованием фигурных скобок (как у вектора `screens`).



При предоставлении внутриклассового инициализатора это следует сделать после знака `=` или в фигурных скобках.

Упражнения раздела 7.3.1

Упражнение 7.23. Напишите собственную версию класса `Screen`.

Упражнение 7.24. Добавьте в свой класс `Screen` три конструктора: стандартный; получающий высоту, ширину и заполняющий содержимое соответствующим количеством пробелов; получающий высоту, ширину и заполняющий символ для содержимого экрана.

Упражнение 7.25. Может ли класс `Screen` безопасно полагаться на заданные по умолчанию версии операторов копирования и присвоения? Если да, то почему? Если нет, то почему?

Упражнение 7.26. Определите функцию `Sales data::avg_price` как встраиваемую.



7.3.2. Функции, возвращающие указатель `*this`

Теперь добавим функции, устанавливающие символ в курсоре или в заданной области:

```
class Screen {
```

```

public:
    Screen & set( char );
    Screen & set( pos, pos, char );
    // другие члены, как прежде
};

inline Screen & Screen::set( char c ) {
    contents[ cursor ] = c; // установите новое значение
    в текущей позиции
                                // курсора
    return *this;           // вернуть этот объект
как 1-значение
}
inline Screen & Screen::set( pos r, pos col, char ch )
{
    contents[ r * width + col ] = ch; // установить
позицию по данному
                                // значению
    return *this; // вернуть этот объект как 1-
значение
}

```

Как и функция `move()`, функция-член `set()` возвращает ссылку на объект, из которого они вызваны (см. раздел 7.1.2). Возвращающие ссылку функции являются 1-значениями (см. раздел 6.3.2), а это означает, что они возвращают сам объект, а не его копию. Это позволяет связать несколько их вызовов в одно выражение:

```

// переместить курсор в указанную позицию и
присвоить
// символу значение
myScreen.move( 4, 0 ).set( '#' );

```

Эти операции выполняются для того же объекта. В этом выражении сначала перемещается курсор (`move()`) в окно (`myScreen`), а затем устанавливается (`set()`) заданный символ. Таким образом, этот оператор эквивалентен следующему:

```

myScreen.move( 4, 0 );
myScreen.set( '#' );

```

Если бы функции `move()` и `set()` возвращали тип `Screen`, а не `Screen&`, этот оператор выполнялся бы совсем по-другому. В данном случае он был бы эквивалентен следующему:

```

// если move возвращает Screen, а не Screen&
Screen temp = myScreen.move(4, 0); // возвращаемое
значение было
                                                // бы скопировано
temp.set('#'); // содержимое myScreen осталось бы
неизменно

```

Если бы функция `move()` имела возвращаемое значение не ссылочного типа, то оно было бы копией `*this` (см. раздел 6.3.2). Вызов функции `set()` изменил бы лишь временную копию, а не сам объект `myScreen`.

Возвращение `*this` из константной функции-члена

Теперь добавим функцию `display()`, выводящую содержимое окна. Необходима возможность включать эту операцию в последовательность операций `set()` и `move()`. Поэтому, подобно функциям `set()` и `move()`, функция `display()` возвратит ссылку на объект, для которого она выполняется.

Логически отображение объекта класса `Screen` (окна) не изменяет его, поэтому функцию `display()` следует сделать константным членом. Но если функция `display()` будет константной, то `this` будет указателем на константу, а значение `*this` — константным объектом. Следовательно, типом возвращаемого значения функции `display()` будет `const Screen&`. Однако, если функция `display()` возвратит ссылку на константу, мы не сможем вставить вызов функции `display()` в последовательность действий:

```

Screen myScreen;
// если display возвращает константную ссылку,
// вызов в последовательности будет ошибкой
myScreen.display(cout).set('*');

```

Хотя объект `myScreen` неконстантный, вызов функции `set()` не будет компилироваться. Проблема в том, что константная версия функции `display()` возвращает ссылку на константу, и мы не можем вызвать функцию `set()` для константного объекта.



Тип возвращаемого значения константной функции-члена, возвращающей `*this` как ссылку, должен быть ссылкой на константу.

Перегрузка на основании константности

Функции-члены вполне можно перегружать исходя из того, являются ли они константными или нет, причем по тем же причинам, по которым функцию можно перегружать исходя из того, является ли ее параметр указателем на константу (см. раздел 6.4). Неконстантная версия неприменима для константных объектов; она применима только для константных объектов. Для неконстантного объекта можно вызвать любую версию, но неконстантная версия будет лучшим соответствием.

В этом примере определим закрытую функцию-член `do_display()` для фактического вывода окна. Каждая из функций `display()` вызовет эту функцию, а затем возвратит объект, для которого она выполняется:

```
class Screen {  
public:  
    // display перегружена на основании того, является  
    // ли  
    // объект константой или нет  
    Screen &display( std::ostream &os)  
    { do_display(os); return *this; }  
    const Screen &display( std::ostream &os) const  
    { do_display(os); return *this; }  
private:  
    // функция отображения окна  
    void do_display( std::ostream &os) const { os <<  
contents; }  
    // другие члены как прежде  
};
```

Как и в любом другом случае, при вызове одной функции-члена другой неявно передается указатель `this`. Таким образом, когда функция `display()` вызывает функцию-член `do_display()`, ей неявно передается собственный указатель `this`. Когда неконстантная версия функции `display()` вызывает функцию `do_display()`, ее указатель `this` неявно преобразуется из указателя на неконстанту в указатель на константу (см. раздел 4.11.2).

Когда функция `do_display()` завершает работу, функция `display()` возвращает объект, с которым они работают, обращаясь к значению указателя `this`. В неконстантной версии указатель `this` указывает на неконстантный объект, так что эта версия функции `display()` возвращает обычную, неконстантную ссылку; константная

версия возвращает ссылку на константу.

Когда происходит вызов функции `display()` для объекта, вызываемую версию определяет его константность:

```
Screen myScreen( 5, 3 );
const Screen blank( 5, 3 );
myScreen.set('#').display( cout );           // вызов
неконстантной версии
blank.display( cout );                      // вызов
константной версии
```

Совет. Используйте закрытые вспомогательные функции

Некоторые читатели могут удивиться: зачем дополнительно создавать отдельную функцию `do_display()`? В конце концов, обращение к функции `do_display()` не намного проще, чем осуществляющееся в ней действие.

Зачем же она нужна? Причин здесь несколько.

- Всегда желательно избегать нескольких экземпляров одного кода.
- По мере развития класса функция `display()` может стать значительно более сложной, а следовательно, преимущества одной, а не нескольких копий кода станут более очевидными.
 - Во время разработки в тело функции `display()`, вероятно, придется добавить отладочный код, который в финальной версии будет удален. Это будет проще сделать в случае, когда весь отладочный код находится в одной функции `do_display()`.
 - Поскольку функция `do_display()` объявлена встраиваемой (`inline`), при создании исполняемого кода компилятор и так вставит ее содержимое по месту вызова, поэтому вызов функции не повлечет за собой никаких потерь времени и ресурсов.

Обычно в хорошо спроектированных программах на языке C++ присутствует множество маленьких функций, таких как `do_display()`, которые выполняют всю основную работу, когда их использует набор других функций.

Упражнения раздела 7.3.2

Упражнение 7.27. Добавьте функции `move()`, `set()` и `display()` в свою версию класса `Screen`. Проверьте свой класс, выполнив следующий код:

```
Screen myScreen( 5, 5, 'X' );
```

```
myScreen.move( 4, 0 ).set( '#' ).display( cout );
cout << "\n";
myScreen.display( cout );
cout << "\n";
```

Упражнение 7.28. Что если бы в предыдущем упражнении типом возвращаемого значения функций `move()`, `set()` и `display()` был `Screen`, а не `Screen&`?

Упражнение 7.29. Пересмотрите свой класс `Screen` так, чтобы функции `move()`, `set()` и `display()` возвращали тип `Screen`, а затем проверьте свое предположение из предыдущего упражнения.

Упражнение 7.30. Обращение к членам класса при помощи указателя `this` вполне допустимо, но избыточно. Обсудите преимущества и недостатки явного использования указателя `this` для доступа к членам.

7.3.3. Типы классов

Каждый класс определяет уникальный тип. Два различных класса определяют два разных типа, даже если их члены совпадают. Например:

```
struct First {
    int memi;
    int getMem();
};

struct Second {
    int memi;
    int getMem();
};

First obj1;
Second obj2 = obj1; // ошибка: obj1 и obj2 имеют
разные типы
```



Даже если у двух классов полностью совпадает список членов, они являются разными типами. Члены каждого класса отличны от членов любого другого класса (или любой другой области видимости).

К типу класса можно обратиться непосредственно, используя имя класса как имя типа. В качестве альтернативы можно использовать имя

класса после ключевого слова `class` или `struct`:

```
Sales_data item1; // инициализация значением  
по умолчанию объекта
```

```
// типа Sales_data
```

```
class Sales_data item1; // эквивалентное объявление
```

Оба способа обращения к типу класса эквивалентны. Второй метод унаследован от языка С и все еще допустим в C++.

Объявления класса

Подобно тому, как можно объявить функцию без ее определения (см. раздел 6.1.2), можно *объявить* (class declaration) класс, не определяя его:

```
class Screen; // объявление класса Screen
```

Такое объявление иногда называют предварительным объявлением (forward declaration), оно вводит имя `Screen` в программу и указывает, что оно относится к типу класса. После объявления, но до определения, тип `Screen` считается *незавершенным типом* (incomplete type), т.е. известно, что `Screen` — это тип класса, но не известно, какие члены он содержит.

Использование незавершенного типа весьма ограниченно. Его можно использовать только для определения указателей или ссылок, а также для объявления (но не определения) функций, которые используют этот тип как параметр или тип возвращаемого значения.

Прежде чем можно будет писать код, создающий объекты некоторого класса, его следует определить, а не только объявить. В противном случае компилятор не будет знать, в каком объеме памяти нуждаются его объекты. Аналогично класс должен быть уже определен перед использованием ссылки или указателя для доступа к члену класса. В конце концов, если класс не был определен, компилятор не сможет узнать, какие члены имеет класс.

За одним исключением, рассматриваемым в разделе 7.6, переменные-члены могут быть определены как имеющие тип класса, только если класс был определен. Тип следует завершить, поскольку компилятор должен знать объем памяти, необходимый для хранения переменных-членов. Пока класс не определен, пока его тело не создано, у класса не может быть переменных-членов его собственного типа. Однако класс считается объявленным (но еще не определенным), как только его имя стало видимо. Поэтому у класса могут быть переменные-члены, являющиеся указателями или ссылками на ее собственный тип:

```
class Link_screen {  
    Screen window;
```

```
Link_screen *next;
Link_screen *prev;
};
```

Упражнения раздела 7.3.3

Упражнение 7.31. Определите два класса, X и Y, у которых класс X имеет указатель на класс Y, а Y содержит объект типа X.

7.3.4. Снова о дружественных отношениях

Наш класс Sales_data определил три обычных функции, не являющиеся членом класса, как дружественные (см. раздел 7.2.1). Класс может также сделать дружественным другой класс или объявить дружественными определенные функции-члены другого (определенного ранее) класса. Кроме того, дружественная функция может быть определена в теле класса. Такие функции неявно являются встраиваемыми.

Дружественные отношения между классами

В качестве примера дружественных классов рассмотрим класс Window_mgr (см. раздел 7.3.1), его членам понадобится доступ к внутренним данным объектов класса Screen, которыми они управляют. Предположим, например, что в класс Window_mgr необходимо добавить функцию-член clear(), заполняющую содержимое определенного окна пробелами. Для этого функции clear() нужен доступ к закрытым переменным-членам класса Screen. Для этого класс Screen должен объявить класс Window_mgr дружественным:

```
class Screen {
    // члены класса Window_Mgr смогут обращаться к
закрытым
    // членам класса Screen
    friend class Window_mgr;
    // ... остальное, как раньше в классе Screen
};
```

Функции-члены дружественного класса могут обращаться ко всем членам класса, объявившего его другом, включая не открытые члены. Теперь, когда класс Window_mgr является другом класса Screen, функцию-член clear() класса Window_mgr можно переписать следующим образом:

```
class Window_mgr {
```

```

public:
    // идентификатор области для каждого окна на
    // экране
    using ScreenIndex = std::size_type;
    // сбросить данное окно, заполнив его пробелами
    void clear( ScreenIndex );
private:
    std::vector<Screen> screens{ Screen( 24, 80, ' ' ) };
    void Window_mgr::clear( ScreenIndex i ) {
        // s - ссылка на окно, которое предстоит очистить
        Screen &s = screens[ i ];
        // сбросить данное окно, заполнив его пробелами
        s.contents = string( s.height * s.width, ' ' );
    }
}

```

Сначала определим `s` как ссылку на класс `Screen` в позиции `i` вектора окон. Затем переменные-члены `height` и `width` данного объекта класса `Screen` используются для вычисления количества символов новой строки, содержащей пробелы. Эта заполненная пробелами строка присваивается переменной-члену `contents`.

Если бы функция `clear()` не была дружественной классу `Screen`, то этот код не компилировался бы. Функция `clear()` не смогла бы использовать переменные-члены `height`, `width` или `contents` класса `Screen`. Поскольку класс `Screen` установил дружественные отношения с классом `Window_mgr`, для его функций доступны все члены класса `Screen`.

Важно понимать, что дружественные отношения не передаются. Таким образом, если у класса `Window_mgr` есть собственные друзья, то у них нет привилегий доступа к членам класса `Screen`.



Каждый класс сам контролирует, какие классы или функции будут его друзьями.

Как сделать функцию-член дружественной

Вместо того чтобы делать весь класс `Window_mgr` дружественным классу `Screen`, можно предоставить доступ только функции-члену `clear()`. Когда функция-член объявляется дружественной, следует указать класс, которому она принадлежит:

```
class Screen {  
    // класс Window_mgr::clear должен быть объявлен  
    // перед классом Screen  
    friend void Window_mgr::clear( ScreenIndex );  
    // ... остальное как раньше в классе Screen  
};
```

Создание дружественных функций-членов требует тщательного структурирования программ в соответствии с взаимозависимостями объявлений и определений. В данном случае программу следует упорядочить следующим образом.

- Сначала определите класс `Window_mgr`, который объявляет, но не может определить функцию `clear()`. Класс `Screen` должен быть объявлен до того, как функция `clear()` сможет использовать члены класса `Screen`.
- Затем определите класс `Screen`, включая объявление функции `clear()` дружественной.
- И наконец, определите функцию `clear()`, способную теперь обращаться к членам класса `Screen`.

Перегруженные функции и дружественные отношения

Хотя у перегруженных функций одинаковое имя, это все же разные функции. Поэтому класс должен объявить дружественной каждую из перегруженных функций:

```
// перегруженные функции storeOn  
extern std::ostream& storeOn( std::ostream &, Screen & );  
  
extern BitMap& storeOn( BitMap &, Screen & );  
class Screen {  
    // версия ostream функции storeOn может обращаться  
    // к закрытым членам  
    // объектов класса Screen  
    friend std::ostream& storeOn( std::ostream &, Screen & ); // ...  
};
```

Класс Screen объявляет другом версию функции storeOn, получающей поток ostream&. Версия, получающая параметр BitMap&, особых прав доступа к объектам класса Screen не имеет.



Объявление дружественных отношений и область видимости

Классы и функции, не являющиеся членами класса, не следует объявлять прежде, чем они будут использованы в объявлении дружественными. Когда имя впервые появляется в объявлении дружественной, оно неявно подразумевается принадлежащей окружающей области видимости. Однако сам друг фактически не объявлен в этой области видимости (см. раздел 7.2.1).

Даже если мы определим функцию в классе, ее все равно придется объявить за пределами класса, чтобы сделать видимой. Объявление должно уже существовать, даже если вызывается дружественная функция:

```
struct X {  
    friend void f() { /* дружественная функция может  
быть определена  
        в теле класса */ }  
    X() { f(); } // ошибка: нет объявления для f  
    void g();  
    void h();  
};  
void X::g() { return f(); } // ошибка: f не была  
объявлена  
void f(); // объявляет функцию,  
определенную в X  
void X::h() { return f(); } // ok: объявление f  
теперь в области  
// видимости
```

Важно понимать, что объявление дружественной затрагивает доступ, но не является объявлением в обычном смысле.



Помните: некоторые компиляторы не выполняют правил поиска имен

друзей (см. раздел 7.2.1).

Упражнения раздела 7.3.4

Упражнение 7.32. Определите собственные версии классов Screen и Window_mgr, в которых функция clear() является членом класса Window_mgr и другом класса Screen.



7.4. Область видимости класса

Каждый класс определяет собственную область видимости. Вне *области видимости класса* (class scope) к обычным данным и функциям его члены могут обращаться только через объект, ссылку или указатель, используя оператор доступа к члену (см. раздел 4.6). Для доступа к членам типа из класса используется оператор области видимости. В любом случае следующее за оператором имя должно быть членом соответствующего класса.

```
Screen::pos ht = 24, wd = 80; // использование типа
pos, определенного
                                         // в классе Screen
Screen scr( ht, wd, ' ' );
Screen *p = &scr;
char c = scr.get(); // доступ к члену get() объекта
scr c = p->get(); // доступ к члену get() из
объекта, на который
                                         // указывает p
```

Область видимости и члены, определенные вне класса

Тот факт, что класс определяет область видимости, объясняет, почему следует предоставить имя класса наравне с именем функции, при определении функции-члена вне ее класса (см. раздел 7.1.2). За пределами класса имена ее членов скрыты.

Как только имя класса становится видимо, остальная часть определения, включая список параметров и тело функции, находится в области видимости класса. В результате мы можем обращаться к другим членам класса без уточнения.

Вернемся, например, к функции-члену `clear()` класса `Window_mgr` (см. раздел 7.3.4). Параметр этой функции имеет тип, определенный в классе `Window_mgr`:

```
void Window_mgr::clear(ScreenIndex i) {
    Screen &s = screens[ i ];
    s.contents = string( s.height * s.width, ' ' );
}
```

Поскольку компилятор видит последующий список параметров и ничего подобного в области видимости класса `WindowMgr`, нет никакой необходимости определять, что требуется тип `ScreenIndex`, определенный в классе `WindowMgr`. По той же причине использование объекта `screens` в теле функции относится к имени, объявленному в классе `Window_mgr`.

С другой стороны, тип возвращаемого значения функции обычно располагается перед именем функции. Когда функция-член определяется вне тела класса, любое имя, используемое в типе возвращаемого значения, находится вне области видимости класса. В результате тип возвращаемого значения должен определять класс, членом которого он является. Например, мы могли бы добавить в класс `Window_mgr` функцию `addScreen()`, добавляющую еще одно окно на экран. Этот член класса возвратит значение типа `ScreenIndex`, которое пользователь впоследствии сможет использовать для поиска этого окна:

```
class Window_mgr {  
public:  
    // добавить окно на экран и возвратить его индекс  
    ScreenIndex addScreen( const Screen& );  
    // другие члены, как прежде  
};  
// тип возвращаемого значения видим прежде, чем  
начинается область  
// видимости класса Window_mgr  
Window_mgr::ScreenIndex  
Window_mgr::addScreen( const Screen &s ) {  
    screens.push_back( s );  
    return screens.size() - 1;  
}
```

Поскольку тип возвращаемого значения встречается прежде имени класса, оно находится вне области видимости класса `Window_mgr`. Чтобы использовать тип `ScreenIndex` для возвращаемого значения, следует определить класс, в котором определяется этот тип.

Упражнения раздела 7.4

Упражнение 7.33. Что будет, если добавить в класс `Screen` переменную-член `size()`, определенную следующим образом? Исправьте все обнаруженные ошибки.

```
pos Screen::size( ) const {
    return height * width;
}
```



7.4.1. Поиск имен в области видимости класса

В рассмотренных до сих пор программах *поиск имен* (name lookup) (процесс поиска объявления, соответствующего данному имени) был относительно прост.

- Сначала поиск объявления осуществляется в том блоке кода, в котором используется имя. Причем рассматриваются только те имена, объявления которых расположены перед местом применения.
- Если имя не найдено, поиск продолжается в иерархии областей видимости, начиная с текущей.
- Если объявление так и не найдено, происходит ошибка.

Когда поиск имен осуществляется в функциях-членах, определенных в классе, может показаться, что он происходит не по правилам поиска. Но в данном случае внешний вид обманчив. Обработка определений классов осуществляется в два этапа.

- Сначала компилируются объявления членов класса.
- Тела функций компилируются только после того, как виден весь класс.



Определения функций-членов обрабатываются *после того*, как компилятор обработает все объявления в классе.

Классы обрабатываются в два этапа, чтобы облегчить организацию кода класса. Поскольку тела функций-членов не обрабатываются, пока весь класс не станет видимым, они смогут использовать любое имя, определенное в классе. Если бы определения функций обрабатывались одновременно с объявлениями переменных-членов, то пришлось бы располагать функции-члены так, чтобы они обращались только к тем именам, которые уже видимы.

Поиск имен для объявлений членов класса

Этот двухэтапный процесс применяется только к именам, используемым в теле функции-члена. Имена, используемые в объявлениях,

включая имя типа возвращаемого значения и типов списка параметров, должны стать видимы прежде, чем они будут использованы. Если объявление переменной-члена будет использовать имя, объявление которого еще не видимо в классе, то компилятор будет искать то имя в той области (областиах) видимости, в которой определяется класс. Рассмотрим пример.

```
typedef double Money;
string bal;
class Account {
public:
    Money balance() { return bal; }
private:
    Money bal;
    // ...
};
```

Когда компилятор видит объявление функции `balance()`, он ищет объявление имени `Money` в классе `Account`. Компилятор рассматривает только те объявления в классе `Account`, которые расположены перед использованием имени `Money`. Поскольку его объявление как члена класса не найдено, компилятор ищет имя в окружающей области видимости. В этом примере компилятор найдет определение типа (`typedef`) `Money`. Этот тип будет использоваться и для типа возвращаемого значения функции `balance()`, и как тип переменной-члена `bal`. С другой стороны, тело функции `balance()` обрабатывается только после того, как видимым становится весь класс. Таким образом, оператор `return` в этой функции возвращает переменную-член по имени `bal`, а не строку из внешней области видимости.

Имена типов имеют особенности

Обычно внутренняя область видимости может переопределить имя из внешней области видимости, даже если это имя уже использовалось во внутренней области видимости. Но если член класса использует имя из внешней области видимости и это имя типа, то класс не сможет впоследствии переопределить это имя:

```
typedef double Money;
class Account {
public:
    Money balance() { return bal; } // используется
```

```

имя Money из внешней // область
видимости
private:
    typedef double Money; // ошибка: нельзя
переопределить Money
    Money bal;
    // ...
} ;

```

Следует заметить, что хотя определение типа Money в классе Account использует тот же тип, что и определение во внешней области видимости, этот код все же ошибочен.

Хотя переопределение имени типа является ошибкой, не все компиляторы обнаружат эту ошибку. Некоторые спокойно примут такой код, даже если программа ошибочна.



Определения имен типов обычно располагаются в начале класса. Так, любой член класса, который использует этот тип, будет расположен после определения его имени.

При поиске имен в областях видимости члены класса следуют обычным правилам

Поиск имени, используемого в теле функции-члена, осуществляется следующим образом.

- Сначала поиск объявления имени осуществляется в функции-члене. Как обычно, рассматриваются объявления в теле функции, только предшествующие месту использования имени.

- Если в функции-члене объявление не найдено, поиск продолжается в классе. Просматриваются все члены класса.

- Если объявление имени в классе не найдено, поиск продолжится в области видимости перед определением функции-члена.

Обычно не стоит использовать имя другого члена класса как имя параметра в функции-члене. Но для демонстрации поиска имени нарушим это правило в функции dummy_fcn() :

```
// обратите внимание: это сугубо демонстрационный
код, отражающий
```

```

// плохую практику программирования. Обычно не
стоит использовать
// одинаковое имя для параметра и функции-члена
int height; // определяет имя, впоследствии
используемое в Screen
class Screen {
public:
    typedef std::string::size_type pos;
    void dummy_fcn( pos height) {
        cursor = width * height; // какое имя height
имеется в виду?
    }
private:
    pos cursor = 0;
    pos height = 0, width = 0;
};

```

Когда компилятор обрабатывает выражение умножения в функции `dummy_fcn()`, он ищет имена сначала в пределах данной функции. Параметры функции находятся в области видимости функции. Таким образом, имя `height`, используемое в теле функции `dummy_fcn()`, принадлежит ее параметру.

В данном случае имя `height` параметра скрывает имя `height` переменной-члена класса. Если необходимо переопределить обычные правила поиска, то это можно сделать так:

```

// плохой подход: имена, локальные для функций-
членов, не должны
// скрывать имена переменных-членов класса
void Screen::dummy_fcn( pos height) {
    cursor = width * this->height; // переменная-член
height
    // альтернативный способ указания переменной-члена
    cursor = width * Screen::height; // переменная-
член height
}

```



Несмотря на то что член класса скрыт, его все равно можно использовать.

Достаточно указать его полное имя, включающее имя класса, либо явно применить указатель `this`.

Значительно проще обеспечить доступ к переменной-члену `height`, присвоив параметру другое имя:

```
// хороший подход: не используйте имена переменных-  
членов для  
// параметров или других локальных переменных  
void Screen::dummy_fcn( pos ht) {  
    cursor = width * height; // переменная-член height  
}
```

Теперь, когда компилятор будет искать имя `height`, в функции `dummy_fcn()` он его не найдет. Затем компилятор просмотрит класс `Screen`. Поскольку имя `height` используется в функции-члене `dummy_fcn()`, компилятор просмотрит все объявления членов класса. Несмотря на то что объявление имени `height` расположено после места его использования в функции `dummy_fcn()`, компилятор решает, что оно относится к переменной-члену `height`.

После поиска в области видимости класса продолжается поиск в окружающей области видимости

Если компилятор не находит имя в функции или в области видимости класса, он ищет его в окружающей области видимости. В данном случае имя `height` объявлено во внешней области видимости, перед определением класса `Screen`. Однако объект во внешней области видимости скрывается переменной-членом класса по имени `height`. Если необходимо имя из внешней области видимости, к нему можно обратиться явно, используя оператор области видимости:

```
// плохой подход: не скрывайте необходимые имена,  
которые
```

```
// определены в окружающих областях видимости  
void Screen::dummy_fcn( pos ::height) {  
    cursor = width * ::height; // который height?
```

Глобальный

```
}
```



Несмотря на то что глобальный объект был скрыт, используя оператор области видимости, доступ к нему вполне можно получить.

Поиск имен распространяется по всему файлу, где они были применены

Когда член класса определен вне определения класса, третий этап поиска его имени происходит не только в объявлениях глобальной области видимости, которые расположены непосредственно перед определением класса Screen, но и распространяется на остальные объявления в глобальной области видимости. Рассмотрим пример.

```
int height; // определяет имя, впоследствии
используемое в Screen
class Screen {
public:
    typedef std::string::size_type pos;
    void setHeight( pos );
    pos height = 0; // скрывает объявление height из
внешней
                           // области видимости
};

Screen::pos verify( Screen::pos );
void Screen::setHeight( pos var ) {
    // var: относится к параметру
    // height: относится к члену класса
    // verify: относится к глобальной функции
    height = verify( var );
}
```

Обратите внимание, что объявление глобальной функции `verify()` не видимо перед определением класса Screen. Но третий этап поиска имени включает область видимости, в которой присутствует определение члена класса. В данном примере объявление функции `verify()` расположено перед определением функции `setHeight()`, а потому может использоваться.

Упражнения раздела 7.4.1

Упражнение 7.34. Что произойдет, если поместить определение типа `pos` в последнюю строку класса Screen?

Упражнение 7.35. Объясните код, приведенный ниже. Укажите, какое из определений, Type или initVal, будет использовано для каждого из имен. Если здесь есть ошибки, найдите и исправьте их.

```
typedef string 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();
    return val;
}
```

7.5. Снова о конструкторах

Конструкторы — ключевая часть любого класса C++. Основы конструкторов рассматривались в разделе 7.1.4, а в этом разделе описаны некоторые из дополнительных возможностей конструкторов и подробности материала, приведенного ранее.



7.5.1. Список инициализации конструктора

Когда определяются переменные, они, как правило, инициализируются сразу, а не определяются и присваиваются впоследствии:

```
string foo = "Hello World!"; // определить и
инициализировать
string bar; // инициализация по умолчанию
пустой строкой
bar = "Hello World!"; // присвоение нового значения
переменной bar
```

Аналогичное различие между инициализацией и присвоением относится к переменным-членам объектов. Если не инициализировать переменную-член явно в списке инициализации конструктора, она инициализируется значением по умолчанию прежде, чем выполнится тело конструктора. Например:

```
// допустимый, но не самый лучший способ создания
конструктора
// класса Sales_data: нет инициализатора
конструктора
Sales_data::Sales_data(const string &s,
                      unsigned cnt, double price)
{
    bookNo = s;
    units_sold = cnt;
    revenue = cnt * price;
}
```

Эта версия и исходное определение в разделе 7.1.4 дают одинаковый результат: по завершении конструктора переменные-члены содержат те же значения. Различие в том, что исходная версия *инициализирует* свои

переменные-члены, тогда как эта версия *присваивает* значения им. Насколько существенно это различие, зависит от типа переменной-члена.

Иногда применение списка инициализации конструктора неизбежно

Зачастую, но не всегда, можно игнорировать различие между инициализацией и присвоением значения переменной-члену. Переменные-члены, являющиеся константой или ссылкой, должны быть инициализированы. Аналогично члены класса, для типа которых не определен стандартный конструктор, также следует инициализировать. Например:

```
class ConstRef {  
public:  
    ConstRef( int ii );  
private:  
    int i;  
    const int ci;  
    int &ri;  
};
```

Переменные-члены *ci* и *ri* следует инициализировать как любой другой константный объект или ссылку. В результате отсутствие инициализатора конструктора для этих членов будет ошибкой:

```
// ошибка: ci и ri должны быть инициализированы  
ConstRef::ConstRef( int ii ) { // присвоения:  
    i = ii; // ok  
    ci = ii; // ошибка: нельзя присвоить значение  
// константе  
    ri = i; // ошибка: ri никогда не будет  
инициализирована  
}
```

К тому времени, когда начинает выполняться тело конструктора, инициализация уже завершена. Единственный шанс инициализировать константу или ссылочную переменную-член — в инициализаторе конструктора. Вот правильный способ написания этого конструктора:

```
// ok: явная инициализация констант и ссылок  
ConstRef::ConstRef( int ii ) : i(ii), ci(ii), ri(i)  
{ }
```



Для предоставления значений переменным-членам, являющимся константой, ссылкой или классом, у которого нет стандартного конструктора, использование списка инициализации конструктора *неизбежно*.

Совет. Используйте списки инициализации конструктора

Во многих классах различие между инициализацией и присвоением связано исключительно с вопросом эффективности: зачем инициализировать переменную-член и присваивать ей значение, когда ее достаточно просто инициализировать.

Однако важней эффективности тот факт, что некоторые переменные-члены обязательно должны быть инициализированы. При стандартном использовании инициализаторов конструктора можно избежать неожиданных ошибок компиляции, когда класс обладает членом, требующим наличия списка инициализации.

Порядок инициализации переменных-членов класса

Нет ничего удивительного в том, что каждая переменная-член присутствует в списке инициализации конструктора только один раз. В конце концов, зачем переменной-члену два исходных значения?

Но что на самом деле неожиданно, так это то, что список инициализации конструктора задает только значения, используемые для инициализации переменных-членов, но не определяет порядок, в котором осуществляется инициализация.

Порядок инициализации переменных-членов задает их расположение при определении. Порядок расположения инициализаторов в списке инициализации конструктора не влияет на порядок инициализации.

Порядок инициализации зачастую не имеет значения. Но если одна из переменных-членов инициализируется с учетом значения другой, порядок их инициализации критически важен.

В качестве примера рассмотрим следующий класс:

```
class X {  
    int i;  
    int j;  
public:
```

```
// ошибка: i инициализируется прежде j
X( int val ) : j( val ), i( j ) { }
};
```

В данном случае список инициализации конструктора написан так, чтобы инициализировать переменную-член *j* значением *val*, а затем использовать переменную-член *j* для инициализации переменной-члена *i*. Но переменная-член *i* инициализируется первой. В результате попытка инициализации переменной-члена *i* осуществляется в момент, когда переменная-член *j* еще не имеет значения!

Некоторые компиляторы достаточно интеллектуальны, чтобы распознать опасность и выдать предупреждение о том, что переменные-члены в списке инициализации конструктора расположены в порядке, отличном от порядка их объявления.

Рекомендуем

Элементы списка инициализации конструктора имеет смысл располагать в том же порядке, в котором объявлены переменные-члены. Кроме того, старайтесь по возможности избегать применения одних переменных-членов для инициализации других.

Вообще, можно достаточно просто избежать любых проблем, связанных с порядком выполнения инициализации. Достаточно использовать параметры конструктора вместо переменных-членов объекта. Конструктор класса *X*, например, лучше было бы написать следующим образом:

```
X( int val ) : i( val ), j( val ) { }
```

В этой версии порядок инициализации переменных-членов *i* и *j* не имеет значения.

Аргументы по умолчанию и конструкторы

Действие стандартного конструктора класса *Sales_data* подобно конструктору, получающему один строковый аргумент. Единственное отличие в том, что конструктор, получающий строковый аргумент, использует его для инициализации переменной-члена *bookNo*. Стандартный конструктор (неявно) использует стандартный конструктор типа *string* для инициализации переменной *bookNo*. Эти конструкторы можно переписать как единый конструктор с аргументом по умолчанию

(см. раздел 6.5.1):

```
class Sales_data {  
public:  
    // определить стандартный конструктор как  
получающий строковый  
    // аргумент  
    Sales_data( std::string s = "") : bookNo(s) {}  
    // остальные конструкторы без изменений  
    Sales_data( std::string s, unsigned cnt, double  
rev) :  
        bookNo(s), units_sold(cnt),  
revenue(rev*cnt) {}  
    Sales_data( std::istream &is) { read(is, *this); }  
    // остальные члены, как прежде  
};
```

Эта версия класса предоставляет тот же интерфейс, что и исходный из раздела 7.1.4. Обе версии создают тот же объект, когда никаких аргументов не предоставлено или когда предоставлен один строковый аргумент. Поскольку этот конструктор можно вызвать без аргументов, он считается стандартным конструктором класса.



Конструктор, предоставляющий аргументы по умолчанию для всех своих параметров, также считается стандартным конструктором.

Следует заметить, что, вероятно, не нужно использовать аргументы по умолчанию с конструктором `Sales_data()`, который получает три аргумента. Если пользователь предоставляет не нулевое количество проданных книг, следует также гарантировать, что пользователь предоставит и цену, по которой они были проданы.

Упражнения раздела 7.5.1

Упражнение 7.36. Следующий инициализатор ошибочен. Найдите и исправьте ошибку.

```
struct X {  
    X( int i, int j) : base(i), rem(base % j) {}  
    int rem, base;
```

};

Упражнение 7.37. Используя версию класса Sales_data из этого раздела, определите, какой конструктор используется для инициализации каждой из следующих переменных, а также перечислите значения переменных-членов в каждом объекте:

```
Sales_data first_item(cin);
int main() {
    Sales_data next;
    Sales_data last("9-999-99999-9");
}
```

Упражнение 7.38. Конструктору, получающему аргумент типа `istream&`, можно предоставить объект `cin` как аргумент по умолчанию. Напишите объявление конструктора, использующего объект `cin` как аргумент по умолчанию.

Упражнение 7.39. Допустимо ли для конструктора, получающего строку, и конструктора, получающего тип `istream&`, иметь аргументы по умолчанию? Если нет, то почему?

Упражнение 7.40. Выберите одну из следующих абстракций (или абстракцию по собственному выбору). Определите, какие данные необходимы в классе. Предоставьте соответствующий набор конструкторов. Объясните свои решения.

- (a) Book (b) Date (c) Employee
- (d) Vehicle (e) Object (f) Tree

7.5.2. Делегирующий конструктор



Новый стандарт расширяет использование списков инициализации конструктора, позволяя определять так называемые делегирующие конструкторы (*delegating constructor*). Делегирующий конструктор использует для инициализации другой конструктор своего класса. Он "делегирует" некоторые (или все) свои задачи другому конструктору.

Подобно любому другому конструктору, делегирующий конструктор имеет список инициализации переменных-членов и тело функции. Список инициализации делегирующего конструктора содержит элемент, являющийся именем самого класса. Подобно другим инициализаторам переменных-членов класса, имя класса сопровождается заключенным в

скобки списком аргументов. Список аргументов должен соответствовать другому конструктору в классе.

В качестве примера перепишем класс Sales_data так, чтобы использовать делегирующие конструкторы следующим образом:

```
class Sales_data {
public:
    // неделегирующий конструктор инициализирует члены
    // из соответствующих
    // аргументов
    Sales_data( std::string s, unsigned cnt, double
price):
        bookNo( s ), units_sold( cnt ),
revenue( cnt*price ) { }
    // все другие конструкторы делегируют к другому
    // конструктору
    Sales_data(): Sales_data( "", 0, 0 ) {}
    Sales_data( std::string s ): Sales_data( s, 0, 0 ) {}
    Sales_data( std::istream &is ): Sales_data()
    { read( is, *this ); }
    // другие члены как прежде
}
```

В этой версии класса Sales_data все конструкторы, кроме одного, делегируют свою работу. Первый конструктор получает три аргумента и использует их для инициализации переменных-членов, но ничего другого не делает. В этой версии класса определен стандартный конструктор, использующий для инициализации конструктор на три аргумента. Он также не делает ничего, поэтому его тело пусто. Конструктор, получающий строку, также делегирует работу версии на три аргумента.

Конструктор, получающий объект `istream&`, также делегирует свои действия. Он делегирует их стандартному конструктору, который в свою очередь делегирует их конструктору на три аргумента. Как только эти конструкторы заканчивают свою работу, запускается тело конструктора с аргументом `istream&`. Оно вызывает функцию `read()` для чтения данных из потока `istream`.

Когда конструктор делегирует работу другому конструктору, список инициализации и тело делегированного конструктора выполняются оба. В классе Sales_data тела делегируемых конструкторов пусты. Если бы тела конструкторов содержали код, то он выполнялся бы прежде, чем

управление возвратилось бы к телу делегирующего конструктора.

Упражнения раздела 7.5.2

Упражнение 7.41. Перепишите собственную версию класса `Sales_data`, чтобы использовать делегирующие конструкторы. Добавьте в тело каждого конструктора оператор, выводящий сообщение всякий раз, когда он выполняется. Напишите объявления для создания объекта класса `Sales_data` любыми возможными способами. Изучите вывод и удостоверьтесь, что понимаете порядок выполнения делегирующих конструкторов.

Упражнение 7.42. Вернитесь к классу, написанному для упражнения 7.40 в разделе 7.5.1, и решите, может ли какой-нибудь из его конструкторов использовать делегирование. Если да, то напишите делегирующий конструктор (конструкторы) для своего класса. В противном случае рассмотрите список абстракций и выберите ту, которая, по вашему, использовала бы делегирующий конструктор. Напишите определение класса для этой абстракции.



7.5.3. Роль стандартного конструктора

Стандартный конструктор автоматически используется всякий раз, когда объект инициализируется по умолчанию. Инициализация по умолчанию осуществляется в следующем случае.

- При определении нестатических переменных (см. раздел 2.2.1) или массивов (см. раздел 3.5.1) в области видимости блока без инициализаторов.
- Когда класс, который сам обладает членами типа класса, использует синтезируемый стандартный конструктор (см. раздел 7.1.4).
- Когда переменные-члены типа класса не инициализируются явно в списке инициализации конструктора (см. раздел 7.1.4).

Инициализация значением по умолчанию осуществляется в следующем случае.

- Во время инициализации массива, когда предоставляется меньше инициализаторов, чем элементов массива (см. раздел 3.5.1).
- При определении локального статического объекта без инициализатора (см. раздел 6.1.1).

- Когда явно запрашивается инициализация значением по умолчанию в форме выражения `T()`, где `T` — это имя типа. (Конструктор вектора, получающий один аргумент, чтобы определить размер вектора (см. раздел 3.3.1), использует аргумент этого вида для инициализации значением по умолчанию своего элемента.)

Чтобы использоваться в этих контекстах, у классов должен быть стандартный конструктор. Большинство этих контекстов должно быть вполне очевидным.

Однако значительно менее очевидным может быть влияние на классы, у которых есть переменные-члены без стандартного конструктора:

```
class NoDefault {
public:
    NoDefault( const std::string& );
    // далее дополнительные члены, но нет других
конструкторов
};

struct A { // my_mem является открытой по
умолчанию; см. раздел 1.2
    NoDefault my_mem;
};

A a; // ошибка: невозможен синтезируемый
конструктор для A
struct B {
    B() {} // ошибка: нет инициализатора для
b_member
    NoDefault b_member;
};
```



Рекомендуем

На практике почти всегда имеет смысл предоставлять стандартный конструктор, если определены другие конструкторы.

Применение стандартного конструктора

Следующее объявление объекта `obj` компилируется без проблем. Но при попытке его использования компилятор жалуется на невозможность применения к функции синтаксиса доступа к члену.

```
Sales_data obj(); // ok: но определена функция, а
```

не объект

```
if (obj.isbn() == Primer_5th_ed.isbn()) // ошибка:  
obj - функция
```

Проблема в том, что, несмотря на намерение объявить инициализированный значением по умолчанию объект `obj`, фактически была объявлена функция без параметров, возвращающая объект типа `Sales_data`.

Чтобы правильно определить объект, использующий стандартный конструктор для инициализации, следует убрать пустые круглые скобки:

```
// ok: obj - объект, инициализированный значением  
по умолчанию  
Sales_data obj;
```



Распространенной ошибкой среди новичков в C++ является объявление объекта, инициализированного стандартным конструктором, следующим образом:

```
Sales_data obj(); // упс! Это объявление функции, а  
не создание объекта
```

```
Sales_data obj2; // ok: obj2 - это объект, а не  
функция
```

Упражнения раздела 7.5.3

Упражнение 7.43. Предположим, имеется класс `NoDefault`, у которого есть конструктор, получающий параметр типа `int`, но нет стандартного конструктора. Определите класс `C`, у которого есть переменная-член типа `NoDefault`. Определите стандартный конструктор для класса `C`.

Упражнение 7.44. Допустимо ли следующее объявление? Если нет, то почему?

```
vector<NoDefault> vec( 10 );
```

Упражнение 7.45. Определите вектор, содержащий объекты типа `C` из предыдущего упражнения?

Упражнение 7.46. Которое из следующих утверждений, если таковое имеется, ложно? Почему?

(а) Класс должен предоставить по крайней мере один конструктор.

- (b) Стандартный конструктор — это конструктор с пустым списком параметров.
- (c) Если для класса не нужно никаких значений по умолчанию, то класс не должен предоставлять стандартный конструктор.
- (d) Если класс не определяет стандартный конструктор, компилятор сам создает конструктор, который инициализирует каждую переменную-член значением по умолчанию соответствующего типа.



7.5.4. Неявное преобразование типов класса

Как упоминалось в разделе 4.11, язык C++ автоматически осуществляет преобразование некоторых встроенных типов. Обращалось также внимание на то, что классы тоже могут определять неявные преобразования. Каждый конструктор, который может быть вызван с одним аргументом, определяет неявное преобразование в тип класса. Такие конструкторы иногда упоминают как *конструкторы преобразования* (converting constructor). Определение преобразования из типа класса в другой тип рассматривается в разделе 14.9.



Конструктор, который может быть вызван с одиночным аргументом, вполне позволяет определить неявное преобразование из типа параметра в тип класса.

Конструкторы класса `Sales_data`, получающие строку и объект класса `istream`, оба определяют неявные преобразования из этих типов в тип `Sales_data`. Таким образом, можно использовать тип `string` или `istream` там, где ожидается объект типа `Sales_data`:

```
string null_book = "9-999-99999-9";
// создает временный объект типа Sales_data
// с units_sold и revenue равными 0 и bookNo равным
null_book
item.combine(null_book);
Здесь происходит вызов функции-члена combine() класса
```

`Sales_data` со строковым аргументом. Этот вызов совершенно корректен; компилятор автоматически создаст объект класса `Sales_data` из данной строки. Этот вновь созданный (временный) объект класса `Sales_data` передается функции `combine()`. Поскольку параметр функции `combine()` является ссылкой на константу, этому параметру можно передать временный объект.

Допустимо только одно преобразование типов класса

В разделе 4.11.2 обращалось внимание на то, что компилятор автоматически применит только одно преобразование типов класса. Например, следующий код ошибочен, поскольку он неявно использует два преобразования:

```
// ошибка: требует двух пользовательских
преобразований:
// (1) преобразование "9-999-99999-9" в string
//       (2) преобразование временной строки в
Sales_data
item.combine("9-999-99999-9");
```

Если данный вызов необходим, это можно сделать при явном преобразовании символьной строки в объект класса `string` или в объект класса `Sales_data`:

```
// ok: явное преобразование в string,
// неявное преобразование в Sales_data
item.combine(string("9-999-99999-9"));
// ok: неявное преобразование в string,
// явное преобразование в Sales_data
item.combine(Sales_data("9-999-99999-9"));
```

Преобразования типов класса не всегда полезны

Желательно ли преобразование типа `string` в `Sales_data`, зависит от конкретных обстоятельств. В данном случае это хорошая идея. Стока в переменной `null_book`, вероятнее всего, соответствует несуществующему ISBN.

Преобразование из `istream` в `Sales_data` более проблематично:

```
// использует конструктор istream при создании
объекта для передачи
// функции combine
item.combine(cin);
```

Этот код неявно преобразует объект `cin` в объект класса `Sales_item`. Это преобразование осуществляется тот конструктор класса `Sales_data`, который получает тип `istream`. Этот конструктор создает (временный) объект класса `Sales_data` при чтении со стандартного устройства ввода. Затем этот объект передается функции `same_isbn()`.

Этот объект класса `Sales_item` временный (см. раздел 2.4.1). По завершении функции `combine()` никакого доступа к нему не будет. Фактически создается объект, удаляющийся после того, как его значение добавляется в объект `item`.

Предотвращение неявных преобразований, осуществляемых конструктором

Чтобы предотвратить использование конструктора в контексте, который требует неявного преобразования, достаточно объявить его **явным** (*explicit constructor*) с использованием ключевого слова `explicit`:

```
class Sales_data {  
public:  
    Sales_data() = default;  
    Sales_data( const std::string &s, unsigned n,  
double p):  
        bookNo(s), units_sold(n), revenue(p*n)  
    {}  
    explicit Sales_data( const std::string &s):  
bookNo(s) {}  
    explicit Sales_data(std::istream&); // оставльные  
члены, как прежде  
};
```

Теперь ни один из конструкторов не применим для неявного создания объектов класса `Sales_data`. Ни один из предыдущих способов применения теперь не сработает:

```
item.combine(null_book); // ошибка: конструктор  
string теперь явный  
item.combine(cin); // ошибка: конструктор  
istream теперь явный
```

Ключевое слово `explicit` имеет значение только для тех конструкторов, которые могут быть вызваны с одним аргументом. Конструкторы, требующие большего количества аргументов, не используются для неявного преобразования, поэтому нет никакой

необходимости определять их как `explicit`. Ключевое слово `explicit` используется только в объявлениях конструкторов в классе. В определении вне тела класса его не повторяют.

```
// ошибка: ключевое слово explicit допустимо только для
// объявлений конструкторов в заголовке класса
explicit Sales_data::Sales_data(istream& is) {
    read(is, *this);
}
```

Явные конструкторы применяются только для прямой инициализации

Одним из контекстов, в котором происходит неявное преобразования, является использование формы инициализации копированием (со знаком `=`) (см. раздел 3.2.1). С этой формой инициализации нельзя использовать явный конструктор; придется использовать прямую инициализацию:

```
Sales_data item1(null_book); // ok: прямая
инициализация
// ошибка: с явным конструктором нельзя
использовать форму
// инициализации копированием
Sales_data item2 = null_book;
```



Явный конструктор применим только с прямой формой инициализации (см. раздел 3.2.1). Кроме того, компилятор *не будет* использовать этот конструктор в автоматическом преобразовании.

Применение явных конструкторов для преобразований

Хотя компилятор не будет использовать явный конструктор для неявного преобразования, его можно использовать для преобразования явно:

```
// ok: аргумент - явно созданный объект класса
Sales_data
item.combine(Sales_data(null_book));
// ok: static_cast может использовать явный
```

конструктор

```
item.combine(static_cast<Sales_data>(cin));
```

В первом вызове конструктор `Sales_data()` используется непосредственно. Этот вызов создает временный объект класса `Sales_data`, используя конструктор `Sales_data()`, получающий строку. Во втором вызове используется оператор `static_cast` (см. раздел 4.11.3) для выполнения явного, а не неявного преобразования. В этом вызове оператор `static_cast` использует для создания временного объекта класса `Sales_data` конструктор с параметром типа `istream`.

Библиотечные классы с явными конструкторами

У некоторых библиотечных классов, включая уже использованные ранее, есть конструкторы с одним параметром.

- Конструктор класса `string`, получающий один параметр типа `const char*` (см. раздел 3.2.1), не является явным.
- Конструктор класса `vector`, получающий размер вектора (см. раздел 3.3.1), является явным.

Упражнения раздела 7.5.4

Упражнение 7.47. Объясните, должен ли быть явным конструктор `Sales_data()`, получающий строку. Каковы преимущества объявления конструктора явным? Каковы недостатки?

Упражнение 7.48. С учетом того, что конструктор `Sales_data()` не является явным, какие операции происходят во время следующих определений:

```
string null_isbn("9-999-99999-9");
Sales_data item1(null_isbn);
Sales_data item2("9-999-99999-9");
```

Что будет при явном конструкторе `Sales_data()`?

Упражнение 7.49. Объясните по каждому из следующих трех объявлений функции `combine()`, что происходит при вызове `i.combine(s)`, где `i` — это объект класса `Sales_data`, `s` — строка:

- (a) `Sales_data &combine(Sales_data);`
- (b) `Sales_data &combine(Sales_data&);`
- (c) `Sales_data &combine(const Sales_data&) const;`

Упражнение 7.50. Определите, должен ли какой-либо из конструкторов вашего класса `Person` быть явным.

Упражнение 7.51. Как, по вашему, почему вектор определяет свой

конструктор с одним аргументом как явный, а строка нет?



7.5.5. Агрегатные классы

Агрегатный класс (aggregate class) предоставляет пользователям прямой доступ к своим членам и имеет специальный синтаксис инициализации. Класс считается агрегатным в следующем случае.

- Все его переменные-члены являются открытыми (`public`).
- Он не определяет конструкторы.
- У него нет никаких внутриклассовых инициализаторов (см. раздел 2.6.1).
- У него нет никаких базовых классов или виртуальных функций, связанных с классом средствами, которые рассматриваются в главе 15.

Например, следующий класс является агрегатным:

```
struct Data {  
    int ival;  
    string s;  
};
```

Для инициализации переменных-членов агрегатного класса можно предоставить заключенный в фигурные скобки список инициализаторов для переменных-членов:

```
// val1.ival = 0; val1.s = string("Anna")  
Data val1 = { 0, "Anna" };
```

Инициализаторы должны располагаться в порядке объявления переменных-членов. Таким образом, сначала располагается инициализатор для первой переменной-члена, затем для второй и т.д. Следующий пример ошибочен:

```
// ошибка: нельзя использовать "Anna" для  
инициализации ival или 1024  
// для инициализации s  
Data val2 = { "Anna" , 1024 };
```

Как и при инициализации элементов массива (см. раздел 3.5.1), если в списке инициализаторов меньше элементов, чем переменных-членов класса, последние переменные-члены инициализируются значением по умолчанию. Список инициализаторов не должен содержать больше элементов, чем переменных-членов у класса.

Следует заметить, что у явной инициализации переменных-членов

объекта класса есть три существенных недостатка.

- Она требует, чтобы все переменные-члены были открытыми.
- Налагает дополнительные обязанности по правильной инициализации каждой переменной-члена каждого объекта на пользователя класса (а не на его автора). Такая инициализация утомительна и часто приводит к ошибкам, поскольку достаточно просто забыть инициализатор или предоставить неподходящее значение.
- Если добавляется или удаляется переменная-член, придется изменить все случаи инициализации.

Упражнения раздела 7.5.5

Упражнение 7.52. На примере первой версии класса `Sales_data` из раздела 2.6.1 объясните следующую инициализацию. Найдите и исправьте возможные ошибки.

```
Sales_data item = {"978-0590353403", 25, 15.99};
```



7.5.6. Литеральные классы

В разделе 6.5.2 упоминалось, что параметры и возвращаемое значение функции `constexpr` должны иметь литеральные типы. Кроме арифметических типов, ссылок и указателей, некоторые классы также являются литеральными типами. В отличие от других классов, у классов, являющихся литеральными типами, могут быть функции-члены `constexpr`. Такие функции-члены должны отвечать всем требованиям функций `constexpr`. Эти функции-члены неявно константные (см. раздел 7.1.2).

Агрегатный класс (см. раздел 7.5.5), все переменные-члены которого имеют литеральный тип, является литеральным классом. Неагрегатный класс, соответствующий следующим ограничениям, также является литеральным классом.

- У всех переменных-членов должен быть литеральный тип.
- У класса должен быть по крайней мере один конструктор `constexpr`.
- Если у переменной-члена есть внутриклассовый инициализатор, инициализатор для переменной-члена встроенного типа должен быть константным выражением (см. раздел 2.4.4). Если переменная-член имеет тип класса, инициализатор должен использовать его собственный

конструктор `constexpr`.

- Класс должен использовать заданное по умолчанию определение для своего деструктора — функции-члена класса, удаляющего объекты типа класса (см. раздел 7.1.5).

Конструкторы `constexpr`

Хотя конструкторы не могут быть константными (см. раздел 7.1.4), в литеральном классе они могут быть функциями `constexpr` (см. раздел 6.5.2). Действительно, литеральный класс должен предоставлять по крайней мере один конструктор `constexpr`.



Конструктор `constexpr` может быть объявлен как `= default` (см. раздел 7.1.4) или как удаленная функция, которые будут описаны в разделе 13.1.6. В противном случае конструктор `constexpr` должен отвечать требованиям к конструкторам (у него не может быть оператора `return`) и к функциям `constexpr` (его исполняемый оператор может иметь единственный оператор `return`) (см. раздел 6.5.2). В результате тело конструктора `constexpr` обычно пусто. Определению конструктора `constexpr` предшествует ключевое слово `constexpr`:

```
class Debug {
public:
    constexpr Debug( bool b = true) : hw( b), io( b),
other( b) { }
    constexpr Debug( bool h, bool i, bool o):
        hw( h), io( i), other( o) { }
    constexpr bool any() { return hw || io || other; }
    void set_io( bool b) { io = b; }
    void set_hw( bool b) { hw = b; }
    void set_other( bool b) { other = b; }
private:
    bool hw;           // аппаратная ошибка, отличная от
ошибки IO
    bool io;           // ошибка IO
    bool other;         // другие ошибки
};
```

Конструктор `constexpr` должен инициализировать каждую переменную-член. Инициализаторы должны либо использовать

конструктор `constexpr`, либо быть константным выражением.

Конструктор `constexpr` используется и для создания объектов `constexpr`, и для параметров или типов возвращаемого значения функций `constexpr`:

```
constexpr Debug io_sub(false, true, false); // отладка IO
if (io_sub.any()) // эквивалент if (true)
    cerr << "print appropriate error messages" << endl;
constexpr Debug prod(false); // при выпуске без отладки
if (prod.any()) // эквивалент if (false)
    cerr << "print an error message" << endl;
```

Упражнения раздела 7.5.6

Упражнение 7.53. Определите собственную версию класса `Debug`.

Упражнение 7.54. Должны ли члены класса `Debug`, начинающиеся с `set_`, быть объявлены как `constexpr`? Если нет, то почему?

Упражнение 7.55. Является ли класс `Data` из раздела 7.5.5 литеральным? Если нет, то почему? Если да, то почему он является литеральным.

7.6. Статические члены класса

Иногда классы нуждаются в членах, ассоциированных с самим классом, а не с его индивидуальными объектами. Например, класс банковского счета, возможно, нуждается в переменной-члене, представляющей базовую процентную ставку. В данном случае мы хотели бы ассоциировать процентную ставку с классом, а не с каждым конкретным объектом. С точки зрения эффективности нет никаких причин хранить процентную ставку для каждого объекта. Однако важней всего то, что если процентная ставка изменится, каждый объект сразу использует новое значение.

Объявление статических членов

Чтобы сделать член класса статическим, его объявление следует предварить ключевым словом `static`. Статические члены, как и любые другие, могут быть открытыми или закрытыми. Статическая переменная-член может быть константой, ссылкой, массивом, классом и т.д.

В качестве примера определим класс, представляющий банковскую учетную запись:

```
class Account {  
public:  
    void calculate() { amount += amount *  
interestRate; }  
    static double rate() { return interestRate; }  
    static void rate(double);  
private:  
    std::string owner;  
    double amount;  
    static double interestRate;  
    static double initRate();  
};
```

Статические члены класса существуют вне конкретного объекта. Объекты не содержат данные, связанные со статическими переменными-членами. Таким образом, каждый объект класса `Account` будет содержать две переменные-члена — `owner` и `amount`. Есть только один объект `interestRate`, совместно используемый всеми объектами `Account`.

Аналогично статические функции-члены не связаны с конкретным объектом; у них нет указателя `this`. В результате статические функции-члены не могут быть объявлены константами и к указателю `this` нельзя

обратиться в теле статического члена класса. Это ограничение применимо и к явному использованию указателя `this`, и к неявному, при вызове не статического члена класса.

Использование статических членов класса

К статическому члену класса можно обратиться непосредственно, используя оператор области видимости:

```
double r;  
r = Account::rate(); // доступ к статическому члену  
при помощи  
                    // оператора области видимости
```

Даже при том, что статические члены не являются частью отдельных объектов, для доступа к статическому члену класса можно использовать объект, ссылку или указатель на тип класса:

```
Account ac1;  
Account *ac2 = &ac1;  
// эквивалентные способы вызова статической функции  
rate r = ac1.rate(); // через объект класса Account  
или ссылку  
r = ac2->rate(); // через указатель на объект  
класса Account
```

Функции-члены могут использовать статические члены непосредственно, без оператора области видимости:

```
class Account {  
public:  
    void calculate() { amount += amount *  
interestRate; }  
private:  
    static double interestRate; // остальные члены как  
прежде  
};
```

Определение статических членов

Подобно любой другой функции-члену, статическую функцию-член можно определить как в, так и вне тела класса. Когда статический член класса определяется вне его тела класса, ключевое слово `static` повторять не нужно, оно присутствует только в объявлении в теле класса:

```
void Account::rate(double newRate) {  
    interestRate = newRate;
```

}



При обращении к статическому члену класса вне тела класса, подобно любому другому члену класса, необходимо указать класс, в котором он определен. Но ключевое слово `static` используется *только* при объявлении в теле класса. В определении ключевое слово `static` не используется.

Поскольку статические переменные-члены не принадлежат индивидуальным объектам класса, они не создаются при создании объектов класса. В результате они не инициализируются конструкторами класса. Кроме того, статическую переменную-член вообще нельзя инициализировать в классе. Каждую статическую переменную-член следует определить и инициализировать вне тела класса. Как и любой другой объект, статическая переменная-член может быть определена только однажды.

Как и глобальные объекты (см. раздел 6.1.1), статические переменные-члены определяются вне любой функции. Следовательно, сразу после определения они продолжают существовать, пока программа не завершит работу.

Статические члены определяют точно так же, как и функции-члены класса вне класса. Указывается тип объекта, затем имя класса, оператор области видимости и собственное имя члена:

```
// определить и инициализировать статический член  
класса double
```

```
Account::interestRate = initRate();
```

Этот оператор определяет статический объект по имени `interestRate`, который является членом класса `Account` и имеет тип `double`. Подобно другим членам класса, определение статического находится в области видимости того класса, где определено его имя. В результате статическую функцию-член `initRate()` можно использовать для инициализации переменной `rate` непосредственно, без уточнения класса. Обратите внимание: несмотря на то, что функция-член `initRate()` является закрытой, ее можно использовать для инициализации объекта `interestRate`. Определение переменной-члена

`interestRate`, подобно любому другому определению, находится в области видимости класса, а следовательно, имеет доступ к закрытым членам класса.



Наилучший способ гарантировать, что объект будет определен только один раз, — разместить определение статических переменных-членов в том же файле, который содержит определение не встраиваемых функций-членов класса.

Инициализация статических переменных-членов в классе

Обычно статические переменные-члены не могут быть инициализированы в теле класса. Но можно предоставить внутриклассовые инициализаторы для тех статических переменных-членов, которые имеют тип целочисленных констант, или статических членов `constexpr` литерального типа (см. раздел 7.5.6). Инициализаторы должны быть константными выражениями. Такие члены сами являются константными выражениями; они могут быть использованы там, где ожидается константное выражение. Например, инициализированную статическую переменную-член можно использовать для определения размерности члена типа массива:

```
class Account {  
public:  
    static double rate() { return interestRate; }  
    static void rate(double);  
private:  
    static constexpr int period = 30; // period -  
// константное выражение  
    double daily_tbl[period];  
};
```

Если член класса используется только в контекстах, где компилятор может подставить его значение, то инициализированная константа или статическое константное выражение не следует определять отдельно. Но если член класса используется в контексте, где значение не может быть подставлено, то определение для этого члена необходимо.

Например, если переменная `period` используется только для определения размерности массива `daily_tbl`, нет никакой

необходимости определять ее за пределами класса `Account`. Но если пропустить определение, то даже, казалось бы, тривиальное изменение в программе может привести к отказу компиляции. Например, если передать переменную-член `Account::period` функции, получающей параметр типа `const int&`, то переменную `period` следует определить.

Если инициализатор предоставляется в классе, определение члена класса не должно задавать исходного значения:

```
// определение статического члена без
инициализатора
constexpr int Account::period; // инициализатор
предоставлен в // определении
класса
```

Рекомендуем

Даже если константная статическая переменная-член инициализируется в теле класса, она должна определяться вне определения класса.

Статические члены можно применять так, как нельзя применять обычные

Как уже упоминалось, статические члены существуют независимо от конкретного объекта. В результате они применимы такими способами, которые недопустимы для нестатических переменных-членов. Например, у статической переменной-члена может быть незавершенный тип (см. раздел 7.3.3). В частности, статическая переменная-член может иметь тип, совпадающий с типом класса, членом которого она является. Нестатическая переменная-член может быть только указателем или ссылкой на объект собственного класса:

```
class Bar {
public:
    // ...
private:
    static Bar mem1; // ok: тип статического члена
может быть
    // незавершенным
    Bar *mem2; // ok: тип указателя-члена может
быть незавершенным
```

```
    Bar mem3;           // ошибка: тип переменной-члена
должен быть
                                // завершенным
};
```

Еще одно различие между статическими и обычными членами в том, что статический член можно использовать как аргумент по умолчанию (см. раздел 6.5.1):

```
class Screen {
public:
    // bkground ссылается на статический член класса
    // объявлено позже, в определении класса
    Screen& clear(char = bkground);
private:
    static const char bkground;
};
```

Нестатическая переменная-член не может использоваться как аргумент по умолчанию, поскольку ее значение является частью объекта, которому она принадлежит. Использование нестатической переменной-члена как аргумента, по умолчанию не предоставляющего объект, которому она принадлежит, также является ошибкой.

Упражнения раздела 7.6

Упражнение 7.56. Что такое статический член класса? Каковы преимущества статических членов? Чем они отличаются от обычных членов?

Упражнение 7.57. Напишите собственную версию класса Account.

Упражнение 7.58. Какие из следующих объявлений и определений статических переменных-членов являются ошибочными? Объясните почему.

```
// 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;
```

```
vector<double> Example::vec;
```

Резюме

Классы — это фундаментальный компонент языка C++. Классы позволяют определять новые типы, наилучшим образом приспособленные к задачам конкретного приложения и позволяющие сделать их короче и проще в модификации.

Основой классов являются абстракция данных (способность определять данные и функции-члены) и инкапсуляция (способность защитить члены класса от общего доступа). Инкапсуляция класса достигается определением членов его реализации закрытыми. Классы могут предоставить доступ к своему не открытому члену, объявив другой класс или функцию дружественной.

Классы могут определять конструкторы — специальные функции-члены, контролирующие инициализацию объектов. Конструкторы могут быть перегружены. Для инициализации всех переменных-членов конструкторы должны использовать список инициализации конструктора.

Классы позволяют объявлять переменные-члены изменяемыми (`mutable`) или статическими (`static`). Изменяемая переменная-член никогда не становится константой — ее значение может быть изменено даже в константной функции-члене. Статической может быть как функция, так и переменная-член. Статические члены класса существуют независимо от объектов данного класса.

Классы могут также определить изменяемые (`mutable`) и статические (`static`) члены. Изменяемая переменная-член никогда не становится константой; ее значение может быть изменено даже в константной функции-члене. Статический член может быть функцией или переменной; статические члены существуют независимо от объектов типа класса.

Термины

= `default`. Синтаксис, используемый после списка параметров объявления стандартного конструктора класса, чтобы сообщить компилятору о необходимости создать конструктор, даже если у класса есть другие конструкторы.

Абстрактный тип данных (`abstract data type`). Структура данных, инкапсулирующая (скрывающая) свою реализацию.

Абстракция данных (`data abstraction`). Технология программирования,

сосредоточенная на интерфейсе типа. Абстракция данных позволяет программистам игнорировать детали реализации типа, интересуясь лишь его возможностями. Абстракция данных является основой как объектно-ориентированного, так и обобщенного программирования.

Агрегатный класс (aggregate class). Класс только с открытыми переменными-членами, без внутриклассовых инициализаторов или конструкторов. Члены агрегатного класса могут быть инициализированы заключенным в фигурные скобки списком инициализаторов.

Делегирующий конструктор (delegating constructor). Конструктор со списком инициализации, один элемент которого определяет другой конструктор того же класса для инициализации.

Дружественные отношения (friend). Механизм, при помощи которого класс предоставляет доступ к своим не открытым членам. Дружественные классы и функции имеют те же права доступа, что и члены самого класса. Дружественными могут быть объявлены как классы, так и отдельные функции.

Закрытый член класса (private member). Члены, определенные после спецификатора доступа `private`; доступный только для друзей и других членов класса. Закрытыми обычно объявляют переменные-члены и вспомогательные функции, используемые классом, но не являющиеся частью интерфейса типа.

Изменяемая переменная-член (mutable data member). Переменная-член, которая никогда не становится константой, даже когда является членом константного объекта. Значение изменяемой переменной-члена вполне может быть изменено в константной функции.

Инкапсуляция (encapsulation). Разделение реализации и интерфейса. Инкапсуляция скрывает детали реализации типа. В языке C++ инкапсуляция предотвращает доступ обычного пользователя класса к его закрытым членам.

Интерфейс (interface). Открытые (`public`) операции, поддерживаемые типом. Обычно интерфейс не включает переменные-члены.

Класс (class). Механизм языка C++, позволяющий создавать собственные абстрактные типы данных. Классы могут содержать как данные, так и функции. Класс определяет новый тип и новую область видимости.

Ключевое слово `class`. Следующие после ключевого слова `class` объявления класса считаются по умолчанию закрытыми (`private`).

Ключевое слово `struct`. Следующие после ключевого слова `struct`

объявления структуры считаются по умолчанию открытыми (`public`).

Константная функция-член (`const member function`). Функция-член, которая не может изменять обычные (т.е. нестатические и неизменяемые) переменные-члены объекта. Указатель `this` константного члена класса является указателем на константу. Функция-член может быть перегружена на основании того, является ли она константной или нет.

Конструктор (`constructor`). Специальная функция-член, обычно инициализирующая объекты. Конструктор должен присвоить каждой переменной-члену хорошо продуманное исходное значение.

Конструктор преобразования (`converting constructor`). Неявный конструктор, который может быть вызван с одиночным аргументом. Конструктор преобразования используется для неявного преобразования типа аргумента в тип класса.

Спецификатор доступа (`access specifier`). Ключевые слова `public` и `private` определяют, доступны ли данные члены для пользователей класса или только его друзьям и членам. Спецификаторы могут присутствовать многократно в пределах класса. Каждый спецификатор устанавливает степень доступа для последующих членов до следующего спецификатора.

Незавершенный тип (`incomplete type`). Тип, который уже объявлен, но еще не определен. Использовать незавершенный тип для определения члена класса или переменной нельзя. Однако ссылки или указатели на незавершенные типы вполне допустимы.

Область видимости класса (`class scope`). Каждый класс определяет область видимости. Область видимости класса сложнее, чем другие области видимости, поскольку определенные в теле класса функции-члены могут использовать имена, которые появятся уже после определения.

Объявление класса (`class declaration`). Ключевое слово `class` (или `struct`), сопровождаемое именем класса и точкой с запятой. Если класс объявлен, но не определен, то это незавершенный тип.

Открытый член класса (`public member`). Члены, определенные после спецификатора доступа `public`; доступны для любого пользователя класса. Обычно в разделах `public` определяют только те функции, которые определяют интерфейс класса.

Поиск имени (`name lookup`). Процесс поиска объявления используемого имени.

Предварительное объявление (`forward declaration`). Объявление имени еще не определенного класса. Как правило, используется для ссылки на

объявление класса до его определения. См. незавершенный тип.

Реализация (implementation). Как правило, закрытые (`private`) члены класса, определяющие данные и все операции, которые не предназначены для использования кодом, применяющим тип.

Синтезируемый стандартный конструктор (synthesized default constructor). Компилятор самостоятельно создает (синтезирует) стандартный конструктор для классов, у которых не определено никаких конструкторов. Этот конструктор инициализирует переменные-члены типа класса, запуская их стандартные конструкторы, а переменные-члены встроенных типов остаются неинициализированными.

Список инициализации конструктора (constructor initializer list). Перечень исходных значений переменных-членов класса. Инициализация переменных-членов класса значениями списка осуществляется прежде, чем выполняется тело конструктора. Переменные-члены класса, которые не указаны в списке инициализации, инициализируются неявно, своими значениями по умолчанию.

Стандартный конструктор (default constructor). Конструктор без параметров.

Указатель `this`. Значение, неявно передаваемое как дополнительный аргумент каждой нестатической функции-члену. Указатель `this` указывает на объект, функция которого вызывается.

Функция-член (member function). Член класса, являющийся функцией. Обычные функции-члены связаны с объектом класса при помощи неявного указателя `this`. Статические функции-члены с объектом не связаны и указателя `this` не имеют. Функции-члены вполне могут быть перегружены; если это так, то неявный указатель `this` участвует в подборе функции.

Явный конструктор (explicit constructor). Конструктор с одним аргументом, который, однако, не может быть использован для неявного преобразования. Объявление явного конструктора предваряется ключевым словом `explicit`.

Часть II

Библиотека C++

С каждым выпуском новой версии языка C++ росла также его библиотека. На самом деле библиотеке посвящено больше двух третей текста нового стандарта. Хоть мы и не можем рассмотреть каждое средство библиотеки подробно, ее основные средства каждый программист C++ должен знать. Эти основные средства мы и рассмотрим в данной части.

Начнем в главе 8 с базовых средств библиотеки IO. Кроме потоков чтения и записи, связанных с окном консоли, библиотека определяет типы, позволяющие читать и писать в именованные файлы и строки в оперативной памяти.

Основную часть библиотеки составляют многочисленные классы контейнеров и семейство обобщенных алгоритмов, позволяющих писать компактные и эффективные программы. Чтобы разработчик программы мог сосредоточиться на решении фактических проблем, библиотека берет на себя все подробности управления памятью.

В главе 3 мы познакомились с контейнером типа `vector`. Подробней мы рассмотрим его и другие типы последовательных контейнеров в главе 9, а также изучим больше операций, предоставленных типом `string`. Строку типа `string` можно считать специальным контейнером, который содержит только символы. Тип `string` поддерживает многие, но не все операции контейнеров.

В главе 10 представлены обобщенные алгоритмы. Обычно они работают с диапазоном элементов в последовательном контейнере или с другой последовательностью. Библиотека алгоритмов предоставляет эффективные реализации различных классических алгоритмов, такие как сортировка и поиск, а также другие общие задачи. Например, есть алгоритм `copy`, который копирует элементы из одной последовательности в другую; алгоритм `find`, который ищет указанный элемент; и так далее. Алгоритмы обобщены двумя способами: они могут быть применены к различным видам последовательностей, и эти последовательности могут содержать элементы различных типов.

Библиотека предоставляет также несколько ассоциативных контейнеров, являющихся темой главы 11. Доступ к элементам в ассоциативном контейнере осуществляется по ключу. Ассоциативные контейнеры имеют много общих операций с последовательными

контейнерами, а также определяют операции, являющиеся специфическими для ассоциативных контейнеров.

Завершается часть главой 12, рассматривающей средства управления динамической памятью, предоставляемые языком и библиотекой. В этой главе рассматриваются одни из самых важных новых библиотечных классов, являющихся стандартизованными версиями интеллектуальных указателей. Используя интеллектуальные указатели, можно сделать намного надежней код, который использует динамическую память. Эта глава завершается расширенным примером, в котором используются библиотечные средства, представленные во всей части II.

Глава 8

Библиотека ввода и вывода

Язык C++ не имеет дела с вводом и выводом непосредственно. Вместо этого ввод и вывод обрабатываются семейством типов, определенных в стандартной библиотеке. Они обеспечивают взаимосвязь с устройствами, файлами, окнами и консолью. Есть также типы, обеспечивающие ввод и вывод в оперативную память и строки.

Библиотека ввода и вывода определяет также операции чтения и записи значений встроенных типов. Кроме того, такие классы, как `string`, обычно определяют подобные операции ввода и вывода для работы с объектами данного класса.

В этой главе представлены основные принципы библиотеки IO. В последующих главах рассматриваются дополнительные возможности: создание собственных операторов ввода и вывода (глава 14), контроль формата и осуществление произвольного доступа к файлам (глава 17).

В предыдущих программах использовалось немало средств библиотеки IO, большинство из них было представлено в разделе 1.2.

- Тип `istream` (input stream — поток ввода) обеспечивает операции ввода.
- Тип `ostream` (output stream — поток вывода) обеспечивает операции вывода.
 - Объект `cin` класса `istream` читает данные со стандартного устройства ввода.
 - Объект `cout` класса `ostream` записывает данные на стандартное устройство вывода.
 - Объект `cerr` класса `ostream` записывает данные на стандартное устройство сообщений об ошибке. Объект `cerr`, как правило, используется для сообщений об ошибках в программе.
- Оператор `>>` используется для чтения данных, передаваемых в объект класса `istream`.
- Оператор `<<` используется для записи данных, передаваемых в объект класса `ostream`.
 - Функция `getline()` (см. раздел 3.2.2) получает ссылку на объект класса `istream` и ссылку на объект класса `string`, а затем читает слово из потока ввода в строку.



8.1. Классы ввода-вывода

Типы и объекты ввода-вывода, которые мы использовали до сих пор, манипулируют символьными данными. По умолчанию эти объекты связаны с окном консоли пользователя. Конечно, реальные программы не могут быть ограничены вводом и выводом исключительно в окно консоли. Программам нередко приходится читать или писать в именованные файлы. Кроме того, операции ввода-вывода зачастую удобно использовать для обработки символов в строке. Приложениям также, вероятно, понадобится читать и писать на языках, которые требуют поддержки расширенных символов.

Для поддержки столь разных видов обработки ввода-вывода, кроме уже использованных ранее типов `istream` и `ostream`, библиотека определяет целую коллекцию типов ввода-вывода. Эти типы (табл. 8.1) определены в трех отдельных заголовках: заголовок `iostream` определяет базовые типы, используемые для чтения и записи в поток, заголовок `fstream` определяет типы, используемые для чтения и записи в именованные файлы, заголовок `sstream` определяет типы, используемые для чтения и записи в строки, расположенные в оперативной памяти.

Таблица 8.1. Типы и заголовки библиотеки ввода-вывода

Заголовок	Тип
iostream	<code>istream</code> , <code>wistream</code> — читают данные из потока
	<code>ostream</code> , <code>wostream</code> — записывают данные в поток
	<code>iostream</code> , <code>wiostream</code> — читают и записывают данные в поток
fstream	<code>ifstream</code> , <code>wifstream</code> — читают данные из файла
	<code>ofstream</code> , <code>wofstream</code> — записывают данные в файл
	<code>fstream</code> , <code>wfstream</code> — читают и записывают данные в файл
sstream	<code>istringstream</code> , <code>wistringstream</code> — читают данные из строки
	<code>ostringstream</code> , <code>wostringstream</code> — записывают данные в строку
	<code>stringstream</code> , <code>wstringstream</code> — читают и записывают данные в строку

Для поддержки языков, использующих расширенные символы, библиотека определяет набор типов и объектов, манипулирующих

данными типа `wchar_t` (см. раздел 2.1.1). Имена версий для расширенных символов начинаются с буквы `w`. Например, объекты `wcin`, `wcout` и `wcerr` соответствуют обычным объектам `cin`, `cout` и `cerr`, но для расширенных символов. Такие объекты определяются в том же заголовке, что и типы для обычных символов. Например, заголовок `fstream` определяет типы `ifstream` и `wfstream`.

Взаимоотношения между типами ввода и вывода

Концептуально ни вид устройства, ни размер символов не влияют на операции ввода-вывода. Например, оператор `>>` можно использовать для чтения данных из окна консоли, из файла на диске или из строки. Точно так же этот оператор можно использовать независимо от того, читаются ли символы типа `char` или `wchar_t`.

Используя *наследование* (inheritance), библиотека позволяет игнорировать различия между потоками различных видов. Подобно шаблонам (см. раздел 3.3), связанные наследованием классы можно использовать, не вникая в детали того, как они работают. Более подробная информация о наследовании в языке C++ приведена в главе 15 и в разделе 18.3.

Если говорить коротко, то наследование позволяет сказать, что некий класс происходит от другого класса. Обычно объект производного класса можно использовать так, как будто это объект класса, от которого он происходит.

Типы `ifstream` и `istringstream` происходят от класса `istream`. Таким образом, объект типа `ifstream` или `istringstream` можно использовать так, как будто это объект класса `istream`. Объекты этих типов можно использовать теми же способами, что и объект `cin`. Например, можно вызвать функцию `getline()` объекта `ifstream` или `istringstream` либо использовать их оператор `>>` для чтения данных. Точно так же типы `ofstream` и `ostringstream` происходят от класса `ostream`. Следовательно, объекты этих типов можно использовать теми же способами, что и объект `cout`.



Все, что рассматривается в остальной части этого раздела, одинаково применимо как к простым, файловым и строковым потокам, а также к

потокам для символов типа `char` или `wchar_t`.



8.1.1. Объекты ввода-вывода не допускают копирования и присвоения

Как упоминалось в разделе 7.1.3, объекты ввода-вывода не допускают копирования и присвоения:

```
ofstream out1, out2;  
out1 = out2; // ошибка: нельзя присваивать  
потоковые объекты  
ofstream print(ofstream); // ошибка: нельзя  
инициализировать параметр  
                           // типа ofstream  
out2 = print(out2); // ошибка: нельзя копировать  
потоковые объекты
```

Поскольку объекты типа ввода-вывода нельзя копировать, не может быть параметра или типа возвращаемого значения одного из потоковых типов (см. раздел 6.2.1). Функции, осуществляющие ввод-вывод, получают и возвращают поток через ссылки. Чтение или запись в объект ввода-вывода изменяет его состояние, поэтому ссылка не должна быть константой.

8.1.2. Флаги состояния

В связи с наследованием классов ввода-вывода возможно возникновение ошибок. Некоторые из ошибок исправимы, другие происходят глубоко в системе и не могут быть исправлены в области видимости программы. Классы ввода-вывода определяют функции и флаги, перечисленные в табл. 8.2, позволяющие обращаться к *флагам состояния* (*condition state*) потока и манипулировать ими.

Таблица 8.2. Флаги состояния библиотеки ввода-вывода

<code>strm::iosate</code>	<code>strm</code> — один из типов ввода-вывода, перечисленных в табл. 8.1. <code>iosate</code> — машинно-зависимый целочисленный тип, представляющий флаг состояния потока
<code>strm::badbit</code>	Значение флага <code>strm::iosate</code> указывает, что поток недопустим

<code>strm::failbit</code>	Значение флага <code>strm::iostate</code> указывает, что операция ввода-вывода закончилась неудачей
<code>strm::eofbit</code>	Значение флага <code>strm::iostate</code> указывает, что поток достиг конца файла
<code>strm::goodbit</code>	Значение флага <code>strm::iostate</code> указывает, что поток не находится в недопустимом состоянии. Это значение гарантированно будет нулевым
<code>s.eof()</code>	Возвращает значение <code>true</code> , если для потока <code>s</code> установлен флаг <code>eofbit</code>
<code>s.fail()</code>	Возвращает значение <code>true</code> , если для потока <code>s</code> установлен флаг <code>failbit</code>
<code>s.bad()</code>	Возвращает значение <code>true</code> , если для потока <code>s</code> установлен флаг <code>badbit</code>
<code>s.good()</code>	Возвращает значение <code>true</code> , если поток <code>s</code> находится в допустимом состоянии
<code>s.clear()</code>	Возвращает все флаги потока <code>s</code> в допустимое состояние
<code>s.clear(флаг)</code>	Устанавливает определенный флаг (флаги) потока <code>s</code> в допустимое состояние. Флаг имеет тип <code>strm::iostate</code>
<code>s.setstate(флаг)</code>	Добавляет в поток <code>s</code> определенный флаг. Флаг имеет тип <code>strm::iostate</code>
<code>s.rdstate()</code>	Возвращает текущее состояние потока <code>s</code> как значение типа <code>strm::iostate</code>

В качестве примера ошибки ввода-вывода рассмотрим следующий код:

```
int ival;
cin >> ival;
```

Если со стандартного устройства ввода ввести, например, слово **Вoo**, то операция чтения потерпит неудачу. Оператор ввода ожидал значение типа `int`, но получил вместо этого символ В. В результате объект `cin` перешел в состояние ошибки. Точно так же объект `cin` окажется в состоянии ошибки, если ввести символ конца файла.

Как только произошла ошибка, последующие операции ввода-вывода в этом потоке будут терпеть неудачу. Читать или писать в поток можно только тогда, когда он находится в неошибочном состоянии. Поскольку поток может оказаться в ошибочном состоянии, код должен проверять его, прежде чем использовать. Проще всего определить состояние потокового объекта — это использовать его в условии:

```
while (cin >> word)
// ok: операция чтения успешна ...
```

Условие оператора `while` проверяет состояние потока, возвращаемого выражением `>>`. Если данная операция ввода успешна, состояние остается допустимым и условие выполняется.

Опрос состояния потока

Использование потока в условии позволяет узнать только то, допустим ли он. Это ничего не говорит о случившемся. Иногда необходимо также узнать причину недопустимости потока. Например, действия после достижения конца файла, вероятно, будут отличаться от таковых после ошибки на устройстве ввода-вывода.

Библиотека ввода-вывода определяет машинно-зависимый целочисленный тип `iostate`, используемый для передачи информации о состоянии потока. Этот тип используется как коллекция битов, подобно переменной `quiz1` в разделе 4.8. Классы ввода-вывода определяют четыре значения `constexpr` (разделе 2.4.4) типа `iostate`, представляющие конкретные битовые схемы. Эти значения используются для указания конкретных видов состояний ввода-вывода. Они используются с побитовыми операторами (см. раздел 4.8) для проверки или установки нескольких флагов за раз.

Флаг `badbit` означает отказ системного уровня, такой как неисправимая ошибка при чтении или записи. Как только флаг `badbit` установлен, использовать поток обычно больше невозможно. Флаг `failbit` устанавливается после исправимой ошибки, такой как чтение символа, когда ожидались числовые данные. Как правило, такие проблемы вполне можно исправить и продолжить использовать поток. Достигение конца файла устанавливает флаги `eofbit` и `failbit`. Флаг `goodbit`, у которого гарантированно будет значение 0, не означает отказа в потоке. Если любой из флагов `badbit`, `failbit` или `eofbit` будет установлен, то оценивающее данный поток условие окажется ложным.

Библиотека определяет также набор функций для опроса состояния этих флагов. Функция `good()` возвращает значение `true`, если ни один из флагов ошибок не установлен. Функции `bad()`, `fail()` и `eof()` возвращает значение `true`, когда установлен соответствующий бит. Кроме того, функция `fail()` возвращает значение `true`, если установлен флаг `badbit`. Корректный способ определения общего состояния потока подразумевал бы использование функции `good()` или `fail()`. На самом деле код проверки потока в условии эквивалентен вызову `! fail()`. Функции `bad()` и `eof()` оповещают только о конкретной ошибке.

Управление флагами состояния

Функция-член `rdstate()` возвращает значение типа `iostate`, соответствующее текущему состоянию потока. Функция `setstate()` позволяет установить указанные биты состояния, чтобы указать возникшую проблему. Функция `clear()` перегружена (см. раздел 6.4): одна ее версия не получает никаких аргументов, а вторая получает один аргумент типа `iostate`.

Версия функции `clear()`, не получающая никаких аргументов, сбрасывает все биты отказа. После ее вызова функция `good()` возвращает значение `true`. Эти функции-члены можно использовать следующим образом:

```
// запомнить текущее состояние объекта cin
auto old_state = cin.rdstate();
cin.clear(); // сделать объект cin
допустимым
process_input(cin); // использовать объект cin
cin.setstate(old_state); // вернуть объект cin в
прежнее состояние
```

Версия функции `clear()`, получающая аргумент, ожидает значение типа `iostate`, представляющее новое состояние потока. Для сброса отдельного флага используется функция-член `rdstate()` и побитовые операторы, позволяющие создать новое желаемое состояние.

Например, следующий код сбрасывает биты `failbit` и `badbit`, а бит `eofbit` оставляет неизменным:

```
// сбросить биты failbit и badbit, остальные биты
оставить неизменными
cin.clear(cin.rdstate() & ~cin.failbit &
~cin.badbit);
```

Упражнения раздела 8.1.2

Упражнение 8.1. Напишите функцию, получающую и возвращающую ссылку на объект класса `istream`. Функция должна читать данные из потока до тех пор, пока не будет достигнут конец файла. Функция должна выводить прочитанные данные на стандартное устройство вывода. Перед возвращением потока верните все значения его флагов в допустимое состояние.

Упражнение 8.2. Проверьте созданную функцию, передав ей при вызове объект `cin` в качестве аргумента.

Упражнение 8.3. В каких случаях завершится следующий цикл while?

```
while (cin >> i) /* ... */
```

8.1.3. Управление буфером вывода

Каждый объект ввода-вывода управляет буфером, используемым для хранения данных, которые программа читает или записывает. Например, при выполнении следующего кода литературная строка могла бы быть выведена немедленно или операционная система могла бы сохранить данные в буфере и вывести их позже:

```
os << "please enter a value: ";
```

Использование буфера позволяет операционной системе объединить несколько операций вывода данной программы на системном уровне в одну операцию. Поскольку запись в устройство может занять много времени, возможность операционной системы объединить несколько операций вывода в одну может существенно повысить производительность.

Существует несколько условий, приводящих к сбросу буфера, т.е. к фактической записи на устройство вывода или в файл.

- Программа завершается нормально. Все буфера вывода освобождаются при выходе из функции `main()`.
- В некий случайный момент времени буфер может оказаться заполненным. В этом случае перед записью следующего значения происходит сброс буфера.
- Сброс буфера можно осуществить явно, использовав такой манипулятор, как `endl` (см. раздел 1.2).
- Используя манипулятор `unitbuf`, можно установить такое внутреннее состояние потока, чтобы буфер освобождался после каждой операции вывода. Для объекта `cerr` манипулятор `unitbuf` установлен по умолчанию, поэтому запись в него приводит к немедленному выводу.
- Поток вывода может быть связан с другим потоком. В таком случае буфер привязанного потока сбрасывается при каждом чтении или записи другого потока. По умолчанию объекты `cin` и `cerr` привязаны к объекту `cout`. Следовательно, чтение из потока `cin` или запись в поток `cerr` сбрасывает буфер потока `cout`.

Сброс буфера вывода

В приведенных ранее программах уже не раз использовался

манипулятор `endl`, который записывает символ новой строки и сбрасывает буфер. Существуют еще два подобных манипулятора: `flush` и `ends`. Манипулятор `flush` используется для сброса буфера потока без добавления символов в вывод. Манипулятор `ends` добавляет в буфер нулевой символ, а затем сбрасывает его.

```
cout << "hi!" << endl; // выводит hi, новую строку и сбрасывает буфер
```

```
cout << "hi!" << flush; // выводит hi и сбрасывает буфер, ничего не
```

```
// добавляя
```

```
cout << "hi!" << ends; // выводит hi, нулевой символ
```

```
// и сбрасывает буфер
```

Манипулятор `unitbuf`

Если сброс необходим при каждом выводе, лучше использовать манипулятор `unitbuf`, который сбрасывает буфер потока после каждой записи. Манипулятор `nounitbuf` восстанавливает для потока использование обычного управляемого системой сброса буфера:

```
cout << unitbuf; // при любой записи буфер будет сброшен немедленно
```

```
// любой вывод сбрасывается немедленно, без всякой буферизации
```

```
cout << nounitbuf; // возвращение к обычной буферизации
```

Внимание! При сбое программы буфер не сбрасывается

Буфер вывода *не сбрасывается*, если программа завершается аварийно. При сбое программы вполне вероятно, что выводимые ею данные могут остаться в буфере, ожидая вывода.

При попытке отладить аварийно завершающуюся программу необходимо гарантировать, что любой *подозрительный* вывод будет сброшен сразу. Программист может впустую потратить множество часов на отслеживание вовсе не того кода только потому, что фактически последний буфер вывода просто не сбрасывается.

Связывание потоков ввода и вывода

Когда поток ввода связан с потоком вывода, любая попытка чтения данных из потока ввода приведет к предварительному сбросу буфера,

связанного с потоком вывода. Библиотечные объекты `cout` и `cin` уже связаны, поэтому оператор `cin >> ival;` заставит сбросить буфер, связанный с объектом `cout`.



В интерактивных системах потоки ввода и вывода обычно связаны. При выполнении программы это гарантирует, что приглашения к вводу будут отображены до того, как система перейдет к ожиданию ввода данных пользователем.

Существуют две перегруженные (см. раздел 6.4) версии функции `tie()`: одна не получает никаких аргументов и возвращает указатель на поток вывода, к которому в настоящее время привязан данный объект, если таковой вообще имеется. Функция возвращает пустой указатель, если поток не связан.

Вторая версия функции `tie()` получает указатель на объект класса `ostream` и связывает себя с ним. Таким образом, код `x.tie(&o)` связывает поток `x` с потоком вывода `o`.

Объект класса `istream` или `ostream` можно связать с другим объектом класса `ostream`:

```
cin.tie(&cout); // только для демонстрации:  
библиотека  
// автоматически связывает объекты  
cin и cout  
// old_tie указывает на поток (если он есть),  
// в настоящее время связанный с объектом cin  
ostream *old_tie = cin.tie(nullptr); // объект cin  
больше не связан  
// связь cin и cerr; не лучшая идея, поскольку  
объект cin должен быть  
// привязан к объекту cout  
cin.tie(&cerr); // чтение в cin сбрасывает объект  
cerr, а не cout  
cin.tie(old_tie); // восстановление обычной связи  
между cin и cout
```

Чтобы связать данный поток с новым потоком вывода, функции `tie()`

передают указатель на новый поток. Чтобы разорвать существующую связь, достаточно передать в качестве аргумента значение 0. Каждый поток может быть связан одновременно только с одним потоком. Однако несколько потоков могут связать себя с тем же объектом `ostream`.



8.2. Ввод и вывод в файл

В заголовке `fstream` определены три типа, поддерживающие операции ввода и вывода в файл: класс `ifstream` читает данные из указанного файла, класс `ofstream` записывает данные в файл, класс `fstream` читает и записывает данные в тот же файл. Использование того же файла для ввода и вывода рассматривается в разделе 17.5.3.

Эти типы поддерживают те же операции, что и описанные ранее объекты `cin` и `cout`. В частности, для чтения и записи в файлы можно использовать операторы ввода-вывода (`<<` и `>>`), можно использовать функцию `getline()` (см. раздел 3.2.2) для чтения из потока `ifstream`. Материал, изложенный в разделе 8.1, относится также и к этим типам.

Кроме поведения, унаследованного от типа `iostream`, определенные в заголовке `fstream` типы имеют в дополнение члены для работы с файлами, связанными с потоком. Эти операции перечислены в табл. 8.3, они могут быть вызваны для объектов классов `fstream`, `ifstream` или `ofstream`, но не других типов ввода-вывода.

Таблица 8.3. Операции, специфические для типов заголовка `fstream`

<code>fstream fstrm;</code>	Создает несвязанный файловый поток, <code>fstream</code> — это один из типов, определенных в заголовке <code>fstream</code>
<code>fstream fstrm(s);</code>	Создает объект класса <code>fstream</code> и открывает файл по имени <code>s</code> . Параметр <code>s</code> может иметь тип <code>string</code> или быть указателем на символьную строку в стиле С (см. раздел 3.5.4). Эти конструкторы являются явными (см. раздел 7.5.4). Заданный по умолчанию режим файла зависит от типа <code>fstream</code>
<code>fstream fstrm(s, режим);</code>	Подобен предыдущему конструктору, но открывает файл <code>s</code> в указанном режиме
<code>fstrm.open(s)</code> <code>fstrm.open(s, режим)</code>	Открывает файл <code>s</code> и связывает его с потоком <code>fstrm</code> . Параметр <code>s</code> может иметь тип <code>string</code> или быть указателем на символьную строку в стиле С. Заданный по умолчанию режим файла зависит от типа <code>fstream</code> . Возвращает тип <code>void</code>
<code>fstrm.close()</code>	Закрывает файл, с которым связан поток <code>fstrm</code> . Возвращает тип <code>void</code>

<code>fstrm.is_open()</code>	Возвращает значение типа <code>bool</code> , указывающее, был ли связанный с потоком <code>fstrm</code> файл успешно открыт и не был ли он закрыт
------------------------------	---



8.2.1. Использование объектов файловых потоков

Когда необходимо читать или писать в файл, определяется объект *файлового потока* (*file stream*), который связывается с файлом. Каждый класс файлового потока определяет функцию-член `open()`, которая выполняет все системные операции, необходимые для поиска указанного файла и его открытия для чтения или записи.

При создании файлового потока можно (но не обязательно) указать имя файла. При предоставлении имени файла функция `open()` вызывается автоматически:

```
ifstream in(ifile); // создать объект ifstream и
открыть указанный файл
ofstream out; // файловый поток вывода, не
связанный с файлом
```



Этот код определяет `in` как входной поток, инициализированный для чтения из файла, указанного строковым аргументом `ifile`. Код определяет `out` как поток вывода, который еще не связан с файлом. По новому стандарту имена файлов могут быть переданы как в переменной библиотечного типа `string`, так и в символьном массиве в стиле C (см. раздел 3.5.4). Предыдущие версии библиотеки допускали только символьные массивы в стиле C.

Использование `fstream` вместо `iostream`

Как упоминалось в разделе 8.1, объект производного типа можно использовать в тех местах, где ожидается объект базового типа. Благодаря этому факту функции, получающие ссылку (или указатель) на один из типов `iostream`, могут быть вызваны от имени соответствующего типа `fstream` (или `sstream`). Таким образом, если имеется функция, получающая ссылку `ostream&`, то ее можно вызвать, передав объект типа `ofstream`, то же относится к ссылке `istream&` и типу `ifstream`.

Например, функции `read()` и `print()` (см. раздел 7.1.3) можно использовать для чтения и записи в именованный файл. В этом примере подразумевается, что имена файлов ввода и вывода передаются как аргументы функции `main()` (см. раздел 6.2.5):

```
ifstream input ( argv[ 1 ] ); // открыть файл
транзакций продаж
ofstream output( argv[ 2 ] ); // открыть файл вывода
Sales_data total; // переменная для
хранения текущей суммы
if ( read( input, total ) ) { // прочитать первую
транзакцию
    Sales_data trans; // переменная для
хранения данных следующей
                                // транзакции
    while( read( input, trans ) ) { // читать остальные
транзакции
        if ( total.isbn() == trans.isbn() ) // проверить
isbn
            total.combine( trans ); // обновить текущую сумму
        else {
            print( output, total ) << endl; // отобразить
результат
            total = trans; // обработать следующую книгу
        }
    }
    print ( output, total ) << endl; // отобразить
последнюю транзакцию
} else // ввода нет
    cerr << "No data?!" << endl;
```

Кроме использования именованных файлов, этот код практически идентичен версии программы сложения, приведенной в разделе 7.1.1. Важнейшая часть — вызов функций `read()` и `print()`. Этим функциям можно передать объекты типа `fstream`, хотя типами их параметров определены `istream&` и `ostream&` соответственно.

Функции-члены `open()` и `close()`

Когда определяется пустой объект файлового потока, вызвав функцию `open()`, его впоследствии можно связать с файлом:

```
ifstream in(ifile); // создать объект ifstream и
открыть указанный файл
ofstream out; // файловый поток вывода, не
связанный ни с каким
// файлом
out.open(ifile + ".copy"); // открыть указанный
файл
```

При неудаче вызова функции `open()` устанавливается бит `failbit` (см. раздел 8.1.2). Поскольку вызов функции `open()` может потерпеть неудачу, имеет смысл проверить ее успешность:

```
if (out) // проверить успешность вызова функции
open
    // вызов успешен, файл можно использовать
```

Это подобно использованию объекта `cin` в условии. При неудаче вызова функции `open()` условие не выполняется и мы не будем пытаться использовать объект `in`.

Как только файловый поток будет открыт, он остается связанным с определенным файлом. На самом деле вызов функции `open()` для файлового потока, который уже открыт, приводит к установке бита `failbit`. Последующие попытки использования этого файлового потока потерпят неудачу. Чтобы связать файловый поток с другим файлом, необходимо сначала закрыть существующий файл. Как только файл закрывается, его можно открыть снова:

```
in.close(); // закрыть файл
in.open(ifile + "2"); // открыть другой файл
```

Если вызов функции `open()` успешен, поток устанавливается в такое состояние, что функция `good()` возвратит значение `true`.

Автоматическое создание и удаление

Рассмотрим программу, функция `main()` которой получает список файлов для обработки (см. раздел 6.2.5). У такой программы может быть следующий цикл:

```
// для каждого переданного программе файла
for (auto p = argv + 1; p != argv + argc; ++p) {
    ifstream input(*p); // создает input и открывает
файл
    if (input) { // если ошибки с файлом нет,
обработать его
```

```
    process( input );
} else
    cerr << "couldn't open: " + string( *p );
} // input выходит из области видимости и удаляется
при каждой итерации
```

При каждой итерации создается новый объект класса `ifstream` по имени `input` и открывается файл для чтения. Как обычно, проверяется успех вызова функции `open()`. Если все в порядке, этот файл передается функции, которая будет читать и обрабатывать ввод. В противном случае выводится сообщение об ошибке.

Поскольку объект `input` является локальным для цикла `while`, он создается и удаляется при каждой итерации (см. раздел 5.4.1). Когда объект `fstream` выходит из области видимости, файл, к которому он привязан, автоматически закрывается. На следующей итерации объект `input` создается снова.



Когда объект класса `fstream` удаляется, автоматически вызывается функция `close()`.

Упражнения раздела 8.2.1

Упражнение 8.4. Напишите функцию, которая открывает файл и читает его содержимое в вектор строк, сохраняя каждую строку как отдельный элемент вектора.

Упражнение 8.5. Перепишите предыдущую программу так, чтобы каждое слово сохранялось в отдельном элементе.

Упражнение 8.6. Перепишите программу книжного магазина из раздела 7.1.1 так, чтобы читать транзакции из файла. Передавайте имя файла как аргумент функции `main()` (см. раздел 6.2.5).



8.2.2. Режимы файла

Каждый поток обладает *режимом файла* (file mode), определяющим возможный способ использования файла. Список режимов файла и их

значений приведен в табл. 8.4.

Таблица 8.4. Режимы файла

in	Открывает файл для ввода
out	Открывает файл для вывода
app	Переходит в конец файла перед каждой записью
ate	Переходит в конец файла непосредственно после открытия
trunc	Усекает существующий поток при открытии
binary	Осуществляет операции ввода-вывода в бинарном режиме

Режим файла можно указать при каждом открытии файла, будь то вызов функции `open()` или косвенное открытие файла при инициализации потока именем файла. У режимов, которые можно задать, есть ряд ограничений.

- Режим `out` может быть установлен только для объектов типа `ofstream` или `fstream`.
- Режим `in` может быть установлен только для объектов типа `ifstream` или `fstream`.
- Режим `trunc` может быть установлен, только если устанавливается также режим `out`.
- Режим `app` может быть установлен, только если не установлен режим `trunc`. Если режим `app` установлен, файл всегда открывается в режиме вывода, даже если это не было указано явно.
- По умолчанию файл, открытый в режиме `out`, усекается, даже если не задан режим `trunc`. Чтобы сохранить содержимое файла, открытого в режиме `out`, необходимо либо задать также режим `app`, тогда можно будет писать только в конец файла, либо задать также режим `in`, тогда файл откроется и для ввода, и для вывода. Использование того же файла для ввода и вывода рассматривается в разделе 17.5.3.
- Режимы `ate` и `binary` могут быть установлены для объекта файлового потока любого типа и в комбинации с любыми другими режимами.

Для каждого типа файлового потока задан режим файла по умолчанию, который используется в случае, если режим не задан. Файлы, связанные с потоками типа `ifstream`, открываются в режиме `in`; файлы, связанные с потоками типа `ofstream`, открываются в режиме `out`; а файлы, связанные с потоками типа `fstream`, открываются в режимах `in` и `out`.

Открытие файла в режиме out удаляет существующие данные

По умолчанию при открытии потока типа `ofstream` содержимое файла удаляется. Единственный способ воспрепятствовать удалению данных файла подразумевает установку режима `app`:

```
// file1 усекается в каждом из следующих случаев
ofstream out("file1"); // out и trunc установлены
установлен неявно
ofstream out2("file1", ofstream::out); // trunc
установлен неявно
ofstream out3("file1", ofstream::out |
ofstream::trunc);
// для сохранения содержимого файла следует явно
задать режим app
ofstream app("file2", ofstream::app); // out
установлен неявно
ofstream app2("file2", ofstream::out |
ofstream::app);
```



ВНИМАНИЕ

Единственный способ сохранить существующие данные в файле, открытом потоком типа `ofstream`, — это явно установить режим `app` или `in`.

Режим файла устанавливается при каждом вызове функции open()

Режим файла некоего потока может изменяться при каждом открытии файла.

```
ofstream out; // режим файла не установлен
out.open("scratchpad"); // неявно заданы режимы out
и trunc
out.close(); // out закрыт, его можно использовать
для другого файла
out.open("precious", ofstream::app); // режимы out
и app
out.close();
```

Первый вызов функции `open()` не задает режим вывода явно; этот

файл неявно открывается в режиме `out`. Как обычно, режим `out` подразумевает также режим `trunc`. Поэтому файл `scratchpad`, расположенный в текущем каталоге, будет усечен. Когда открывается файл `precious`, задается режим добавления. Все данные остаются в файле, а запись осуществляется в конец файла.



Режим файла устанавливается при каждом вызове функции `open()` явно или неявно. Когда режим не устанавливается явно, используется значение по умолчанию.

Упражнения раздела 8.2.2

Упражнение 8.7. Пересмотрите программу книжного магазина из предыдущего раздела так, чтобы вывод записывался в файл. Передайте имя этого файла как второй аргумент функции `main()`.

Упражнение 8.8. Пересмотрите программу из предыдущего упражнения так, чтобы добавить ее вывод в заданный файл. Запустите программу для того же выходного файла по крайней мере дважды и удостоверьтесь, что данные сохраняются.

8.3. Строковые потоки

Заголовок `sstream` определяет три типа, поддерживающие операции ввода-вывода в оперативной памяти; эти типы обеспечивают чтение или запись в строку, как будто она является потоком ввода-вывода.

Объект класса `istringstream` читает строку, объект класса `ostringstream` записывает строку, а объект класса `stringstream` читает и записывает строку. Подобно типам заголовка `fstream`, типы, определенные в заголовке `sstream`, происходят от типов, используемых заголовком `iostream`. Кроме унаследованных операций, типы, определенные в заголовке `sstream`, имеют дополнительные члены для работы со строками, связанными с потоком. Эти операции перечислены в табл. 8.5. Они могут быть выбраны для объектов класса `stringstream`, *строковых потоков* (*string stream*), но не других типов ввода-вывода.

Обратите внимание на то, что хотя заголовки `fstream` и `sstream` имеют общий интерфейс к заголовку `iostream`, никакой другой

взаимосвязи у них нет. В частности, нельзя использовать функции `open()` и `close()` для объектов класса `stringstream`, а функцию `str()` нельзя использовать для объектов класса `fstream`.

Таблица 8.5. Операции, специфические для класса `stringstream`

<code>sstream strm;</code>	<code>strm</code> — несвязанный объект класса <code>stringstream</code> . <code>sstream</code> — это один из типов, определенных в заголовке <code>sstream</code>
<code>sstream strm(s);</code>	<code>sstream</code> содержит копию строки <code>s</code> . Этот конструктор является явным (см. раздел 7.5.4).
<code>strm.str()</code>	Возвращает копию строки, которую хранит объект <code>strm</code>
<code>strm.str(s)</code>	Копирует строку <code>s</code> в объект <code>strm</code> . Возвращает тип <code>void</code>

8.3.1. Использование класса `istringstream`

Класс `istringstream` зачастую используют тогда, когда некую работу следует выполнить со всей строкой и другую работу с отдельными словами в пределах строки.

Предположим, например, что имеется файл, содержащий список людей и номеров их телефонов. У одних людей есть только один номер, а у других несколько — домашний телефон, рабочий, мобильный и т.д. Наш исходный файл может выглядеть следующим образом:

```
morgan 2015552368 8625550123
drew 9735550130
lee 6095550132 2015550175 8005550000
```

Каждая запись в этом файле начинается с имени, затем следует один или несколько номеров телефонов. Для начала определим простой класс, представляющий исходные данные:

```
// по умолчанию члены являются открытыми; см.
раздел 1.2
struct PersonInfo {
    string name;
    vector<string> phones;
};
```

Один член класса `PersonInfo` будет представлять имя человека, а вектор будет содержать переменное количество его номеров телефонов.

Наша программа будет читать файл данных и создавать вектор объекта класса `PersonInfo`. Каждый элемент вектора будет соответствовать одной записи в файле. Ввод обрабатывается в цикле, который читает

запись, а затем извлекает имя и номера телефона каждого человека:

```
string line, word; // будут содержать строку и слово из ввода
vector<PersonInfo> people; // будет содержать все записи из ввода
// читать ввод по строке за раз, пока не встретится конец файла
// (или другая ошибка)
while (getline(cin, line)) {
    PersonInfo info; // создать объект для содержания данных записи
    istringstream record(line); // связать запись с читаемой строкой
    record >> info.name; // читать имя
    while (record >> word) // читать номер телефона
        info.phones.push_back(word); // и сохранить их
    people.push_back(info); // добавить эту запись в people
}
```

Здесь для чтения всей записи со стандартного устройства ввода используется функция `getline()`. Если вызов функции `getline()` успешен, то переменная `line` будет содержать запись из входного файла. В цикле `while` определяется локальный объект `PersonInfo` для содержания данных из текущей записи.

Затем с только что прочитанной строкой связывается поток `istringstream`. Теперь для чтения каждого элемента в текущей записи можно использовать оператор ввода класса `istringstream`. Сначала читается имя, затем следует цикл `while`, читающий номера телефонов данного человека.

Внутренний цикл `while` завершается, когда все данные в строке прочитаны. Этот цикл работает аналогично другим, написанным для чтения из объекта `cin`. Различие только в том, что этот цикл читает данные из строки, а не со стандартного устройства ввода. Когда строка прочитана полностью, встретившийся "конец файла" свидетельствует о том, что следующая операция ввода в объект `record` потерпит неудачу.

Внешний цикл `while` завершается добавлением в вектор только что обработанного объекта класса `PersonInfo`. Внешний цикл `while`

продолжается, пока объект `cin` не встретит конец файла.

Упражнения раздела 8.3.1

Упражнение 8.9. Используйте функцию, написанную для первого упражнения 8.1.2, для вывода содержимого объекта класса `istringstream`.

Упражнение 8.10. Напишите программу для сохранения каждой строки из файла в векторе `vector<string>`. Затем используйте объект класса `istringstream` для чтения каждого элемента из вектора по одному слову за раз.

Упражнение 8.11. Программа этого раздела определила свой объект класса `istringstream` во внешнем цикле `while`. Какие изменения необходимо внести, чтобы определить объект `record` вне этого цикла? Перепишите программу, перенеся определение объекта `record` во вне цикла `while`, и убедитесь, все ли необходимые изменения внесены.

Упражнение 8.12. Почему в классе `PersonInfo` не использованы внутриклассовые инициализаторы?

8.3.2. Использование класса `ostringstream`

Класс `ostringstream` полезен тогда, когда необходимо организовать вывод небольшими частями за раз, не откладывая его на более позднее время. Например, могла бы возникнуть необходимость проверять и переформатировать номера телефонов, которые были прочитаны в коде предыдущего примера. Если все номера допустимы, необходимо переформатировать номера и вывести их в новый файл. Если у кого-нибудь будут недопустимые номера, то помещать их в новый файл не нужно. Вместо этого следует вывести сообщение об ошибке, содержащее имя человека и список его недопустимых номеров.

Поскольку нельзя включать для человека данные с недопустимыми номерами, мы не можем произвести вывод, пока не просмотрим и не проверим все их номера. Но можно "записать" вывод в оперативную память объекта класса `ostringstream`:

```
for (const auto &entry : people) { // для каждой записи в people
    ostringstream formatted, badNums; // объекты создаются на каждом
                                         // цикле
    for (const auto &nums : entry.phones) { // для
```

```

каждого номера
    if (!valid(nums)) {
        badNums << " " << nums; // строка в badNums
    } else
        // "запись" в строку formatted
        formatted << " " << format(nums);
    }
    if (badNums.str().empty()) // если плохих номеров
нет
        os << entry.name << " " // вывести имя
            << formatted.str() << endl; // и
переформатированные номера
    else // в противном случае вывести имя и плохие
номера
        cerr << "input error: " << entry.name
            << " invalid number(s) " << badNums.str() <<
endl;
}

```

В этой программе подразумевается, что есть две функции, `valid()` и `format()`, которые проверяют и переформатируют номера телефонов. Интересная часть программы — использование строковых потоков `formatted` и `badNums`. Для записи в эти объекты используется обычный оператор вывода (`<<`). Но они действительно "пишут" строковые манипуляторы. Они добавляют символы к строкам в строковых потоках `formatted` и `badNums` соответственно.

Упражнения раздела 8.3.2

Упражнение 8.13. Перепишите программу номеров телефонов из этого раздела так, чтобы читать из именованного файла, а не из объекта `cin`.

Упражнение 8.14. Почему переменные `entry` и `nums` были объявлены как `const auto &`?

Резюме

Язык C++ использует библиотечные классы для обработки потоков ввода и вывода.

- Класс `iostream` отрабатывает ввод-вывод на консоль.
- Класс `fstream` отрабатывает ввод-вывод в именованным файл.
- Класс `stringstream` отрабатывает ввод-вывод в строки в

оперативной памяти.

Классы `fstream` и `stringstream` связаны происхождением от класса `iostream`. Классы ввода происходят от класса `istream`, а классы вывода — от класса `ostream`. Таким образом, операции, которые могут быть выполнены с объектом класса `istream`, могут быть также выполнены с объектом класса `ifstream` или `istringstream`. Аналогично для классов вывода, происходящих от класса `ostream`.

Каждый объект ввода-вывода обладает набором флагов состояния, указывающих, возможен ли ввод-вывод через этот объект. Если произошла ошибка (например, встретился конец файла в потоке ввода), то состояние объекта окажется таково, что никакой дальнейший ввод невозможен, пока ошибка не будет исправлена. Библиотека предоставляет набор функций для установки и проверки этих состояний.

Термины

Класс `fstream`. Файловый поток, обеспечивающий чтение и запись в тот же файл. По умолчанию объект класса `ifstream` открывает файл одновременно в режимах `in` и `out`.

Класс `ifstream`. Файловый поток, читающий данные из файла. По умолчанию поток `ifstream` открывается в режиме `in`.

Класс `istringstream`. Строковый поток, читающий данные из строки.

Класс `ofstream`. Файловый поток, записывающий данные в файл. По умолчанию поток `ofstream` открывается в режиме `out`.

Класс `ostringstream`. Строковый поток, записывающий данные в строку.

Класс `stringstream`. Строковый поток, читающий и записывающий данные в строку.

Наследование (inheritance). Программное средство, позволяющее типу наследовать интерфейс другого типа. Классы `ifstream` и `istringstream` происходят от классов `istream` и `ofstream`, а класс `ostringstream` происходит от класса `ostream`. Более подробная информация о наследовании приведена в главе 15.

Режим файла (file mode). Флаги классов заголовка `fstream`, устанавливаемые при открытии файла и задающие способ его применения. **Строковый поток (string stream).** Потоковый объект, читающий или записывающий данные в строку. Кроме возможностей, присущих классу

`iostream`, классы строковых потоков определяют перегруженную функцию `str()`. Вызов функции `str()` без аргументов возвращает строку, с которой связан объект строкового потока, а ее вызов со строковым аргументом связывает строковый поток с копией этой строки.

Файловый поток (`file stream`). Потоковый объект этого класса позволяет читать и записывать данные в именованный файл. Кроме возможностей, присущих классу `iostream`, класс `fstream` обладает также функциями-членами `open()` и `close()`. Функция-член `open()` получает символьную строку в стиле C, которая содержит имя открываемого файла и необязательный аргумент, задающий режим. Функция-член `close()` закрывает файл, с которым связан поток. Ее следует вызвать прежде, чем может быть открыт другой файл.

Флаг состояния (`condition state`). Флаги и связанные с ними функции потоковых классов позволяют выяснить, пригоден ли данный поток для использования.

Глава 9

Последовательные контейнеры

Эта глава подробно останавливается на материале главы 3 и завершает обсуждение последовательных контейнеров стандартной библиотеки. Порядок элементов в последовательном контейнере соответствует порядку их добавления в контейнер. В библиотеке определено также несколько ассоциативных контейнеров, позиция элементов которых зависит от ключа, ассоциируемого с каждым элементом. Операции, специфические для ассоциативных контейнеров, рассматриваются в главе 11.

Классы контейнеров имеют общий интерфейс, который каждый из контейнеров дополняет собственным способом. Общий интерфейс упрощает изучение библиотечных классов; то, что стало известно о контейнере одного вида, относится к контейнеру другого. Однако каждый вид контейнеров обладает индивидуальной эффективностью и функциональными возможностями.

Контейнер (*container*) содержит коллекцию объектов определенного типа. *Последовательные контейнеры* (*sequential container*) позволяют контролировать порядок, в котором хранятся элементы и предоставляется доступ к ним. Этот порядок не зависит от значений элементов, он соответствует позиции, в которую помещаются элементы контейнера. В отличие от них, ассоциативные контейнеры (упорядоченные и неупорядоченные) хранят свои элементы на основании значения ключа, как будет описано в главе 11.

Библиотека предоставляет также три контейнерных адаптера, каждый из которых адаптирует определенный тип контейнера, определяя иной интерфейс к функциям контейнера. Адаптеры рассматриваются в конце этой главы.



Эта глава основана на материале разделов 3.2–3.4. Здесь подразумевается, что читатель знаком с их материалом.



9.1. Обзор последовательных контейнеров

Все последовательные контейнеры, перечисленные в табл. 9.1, предоставляют быстрый последовательный доступ к своим элементам. Однако эти контейнеры обладают разной производительностью и возможностями, включая следующие:

- цена добавления и удаления элементов из контейнера;
- цена непоследовательного доступа к элементам контейнера.

Таблица 9.1. Типы последовательных контейнеров

vector	Массив переменного размера (вектор). Обеспечивает быстрый произвольный доступ. Вставка и удаление элементов, кроме как в конец, могут быть продолжительными
deque	Двухсторонняя очередь. Обеспечивает быстрый произвольный доступ. Быстрая вставка и удаление в начало и конец
list	Двухсвязный список. Обеспечивает только двунаправленный последовательный доступ. Быстрая вставка и удаление в любую позицию
forward_list	Односвязный список. Обеспечивает только последовательный доступ в одном направлении. Быстрая вставка и удаление в любую позицию
array	Массив фиксированного размера. Обеспечивает быстрый произвольный доступ. Не позволяет добавлять или удалять элементы
string	Специализированный контейнер, подобный вектору, содержащий символы. Быстрый произвольный доступ. Быстрая вставка и удаление в конец

За исключением контейнера `array`, являющегося контейнером фиксированного размера, контейнеры обеспечивают эффективное, гибкое управление памятью, позволяя добавлять и удалять элементы, увеличивая и сокращая размер контейнера. Стратегии, используемые контейнерами для хранения своих элементов, имеют незначительное, а иногда существенное влияние на эффективность операций с ними. В некоторых случаях эти стратегии влияют также на то, поддерживает ли некий контейнер определенную операцию.

Например, контейнеры `string` и `vector` содержат свои элементы в соседних областях памяти. Поскольку элементы расположены последовательно, их адрес по индексу вычисляется очень быстро. Но добавление или удаление элемента в середину такого контейнера занимает много времени: для обеспечения последовательности все элементы после удаления или вставки придется переместить. Кроме того, добавление

элемента может иногда потребовать резервирования дополнительного места для хранения. В этом случае каждый элемент следует переместить в новое место.

Контейнеры `list` и `forward_list` разработаны так, чтобы быстро добавлять и удалять элементы в любой позиции контейнера. Однако эти типы не поддерживают произвольный доступ к элементам: обратиться к элементу можно, только перебрав контейнер. Кроме того, дополнительные затраты памяти этих контейнеров зачастую являются существенными по сравнению с контейнерами `vector`, `deque` и `array`.

Контейнер `deque` — это более сложная структура данных. Как и контейнеры `string`, `vector` и `deque`, они обеспечивают быстрый произвольный доступ. Как и у контейнеров `string` и `vector`, добавление или удаление элементов в середину контейнера `deque` — потенциально дорогая операция. Однако добавление и удаление элементов в оба конца контейнера `deque` являются быстрой операцией, сопоставимой с таковыми у контейнеров `list` и `forward_list`.



Типы `forward_list` и `array` были добавлены согласно новому стандарту. Класс `array` — более безопасная и легкая в употреблении альтернатива встроенным массивам. Как и встроенные массивы, объект библиотечного класса `array` имеет фиксированный размер. В результате он не поддерживает операции по добавлению и удалению элементов или изменению размеров контейнера. Класс `forward_list` предназначен для замены наилучших самодельных односвязных списков. Следовательно, у него нет функции `size()`, поскольку сохранение или вычисление его размера повлекло бы дополнительные затраты по сравнению с самодельным списком. Для других контейнеров функция `size()` гарантированно будет выполняться быстро с постоянной скоростью.



По причинам, изложенным в разделе 13.6, новые библиотечные контейнеры работают существенно быстрее, чем в предыдущих версиях. Библиотечные контейнеры почти наверняка работают так же (если не лучше), чем даже наиболее тщательно разработанные альтернативы. Современные программы С++ должны использовать библиотечные

контейнеры, а не множество примитивных структур, подобных массивам.

Решение о том, какой последовательный контейнер использовать



Как правило, если нет серьезных оснований предпочесть другой контейнер, лучше использовать контейнер `vector`.

Ниже приведено несколько эмпирических правил по выбору используемого контейнера.

- Если нет причин использовать другой контейнер, используйте вектор.
- Если имеется много небольших элементов и дополнительных затрат, не используйте контейнер `list` или `forward_list`.
- Если нужен произвольный доступ к элементам, используйте контейнер `vector` или `deque`.
- Если необходима вставка или удаление элементов в середину, используйте контейнер `list` или `forward_list`.
- Если необходима вставка или удаление элементов в начало или в конец, но не в середину, используйте контейнер `deque`.
- Если вставка элементов в середину контейнера нужна только во время чтения ввода, а впоследствии нужен произвольный доступ к элементам, то:
 - сначала решите, необходимо ли фактически добавлять элементы в середину контейнера. Зачастую проще добавлять элементы в `vector`, а затем использовать библиотечную функцию `sort()` (рассматриваемую в разделе 10.2.3) для переупорядочивания контейнера по завершении ввода;
 - если вставка в середину необходима, рассмотрите возможность использования контейнера `list` на фазе ввода, а по его завершении — копирования списка в вектор.

Но что если нужен и произвольный доступ, и вставка (удаление) элементов в середину контейнера? Решение будет зависеть от отношения цены доступа к элементам в контейнере `list` и `forward_list` цене вставки (удаления) элементов в контейнер `vector` или `deque`. Как правило, выбор типа контейнера определит преобладающая операция приложения (произвольный доступ или вставка и удаление). В таких случаях, вероятно, потребуется проверка производительности приложения

с использованием контейнеров обоих типов.

Рекомендуем

Если вы не уверены, какой контейнер использовать, напишите свой код так, чтобы он использовал только те операции, которые совпадают у вектора и списка: используйте итераторы, а не индексы, и избегайте произвольного доступа к элементам. Так будет удобней заменить вектор на список при необходимости.

Упражнения раздела 9.1

Упражнение 9.1. Какой из контейнеров (`vector`, `deque` или `list`) лучше подходит для приведенных ниже задач? Объясните, почему. Если нельзя отдать предпочтение тому или иному контейнеру, объясните, почему?

- Чтение известного заранее количества слов и вставка их в контейнер в алфавитном порядке по мере ввода. В следующей главе будет показано, что ассоциативные контейнеры лучше подходят для этой задачи.
- Чтение неизвестного заранее количества слов. Новые слова всегда добавляются в конец. Следующее значение извлекается из начала.
- Чтение неизвестного заранее количества целых чисел из файла. Числа сортируются, а затем выводятся на стандартное устройство вывода.



9.2. Обзор библиотечных контейнеров

Возможные операции с контейнерами составляют своего рода иерархию.

- Некоторые функциональные возможности (табл. 9.2) поддерживаются контейнерами всех типов.
- Другие операции являются специфическими только для последовательных (табл. 9.3), ассоциативных (табл. 11.7) или неупорядоченных (табл. 11.8) контейнеров.
- Остальные являются общими лишь для небольшого подмножества контейнеров.

Таблица 9.2. Средства контейнеров

Псевдонимы типов	
<code>iterator</code>	Тип итератора для контейнера данного типа
<code>const_iterator</code>	Тип итератора, позволяющий читать, но не изменять значение элемента
<code>size_type</code>	Целочисленный беззнаковый тип, размер которого достаточно велик, чтобы содержать значение размера наибольшего возможного контейнера данного типа
<code>difference_type</code>	Целочисленный знаковый тип, размер которого достаточно велик, чтобы содержать значение разницы между двумя итераторами
<code>value_type</code>	Тип элемента
<code>reference</code>	Тип l-значения элемента; то же, что и <code>value_type&</code>
<code>const_reference</code>	Тип константного l-значения элемента; аналог <code>const value_type&</code>
Конструкторы	
<code>C c;</code>	Стандартный конструктор, создающий пустой контейнер
<code>C c1(c2);</code>	Создает контейнер <code>c1</code> как копию контейнера <code>c2</code>
<code>C c(b, e);</code>	Копирует элементы из диапазона, обозначенного итераторами <code>b</code> и <code>e</code> (недопустимо для массива)
<code>C c{ a, b, c...};</code>	Списочная инициализация контейнера <code>c</code>
Присвоение и замена	
<code>c1 = c2</code>	Заменяет элементы контейнера <code>c1</code> элементами

	контейнера <code>c2</code>
<code>c1 = { a, b, c... }</code>	Заменяет элементы контейнера <code>c1</code> элементами списка <i>(недопустимо для массива)</i>
<code>a. swap(b)</code>	Меняет местами элементы контейнеров <code>a</code> и <code>b</code>
<code>swap(a, b)</code>	Эквивалент <code>a. swap(b)</code>
Размер	
<code>c.size()</code>	Возвращает количество элементов контейнера <code>c</code> <i>(недопустимо для контейнера <code>forward_list</code>)</i>
<code>c.max_size()</code>	Возвращает максимально возможное количество элементов контейнера <code>c</code>
<code>c.empty()</code>	Возвращает логическое значение <code>false</code> , если контейнер <code>c</code> пуст. В противном случае возвращает значение <code>true</code>
Добавление/удаление элементов (недопустимо для массива)	Примечание: интерфейс этих функций зависит от типа контейнера
<code>c.insert(args)</code>	Копирует элемент(ы), указанный параметром <code>args</code> , в контейнер <code>c</code>
<code>c.emplace(inits)</code>	Использует параметр <code>inits</code> для создания элемента в контейнере <code>c</code>
<code>c.erase(args)</code>	Удаляет элемент(ы), указанный параметром <code>args</code> , из контейнера <code>c</code>
<code>c.clear()</code>	Удаляет все элементы из контейнера <code>c</code> ; возвращает значение <code>void</code>
Операторы равенства и отношения	
<code>==, !=</code>	Равенство допустимо для контейнеров всех типов
<code><, <=, >, >=</code>	Операторы отношения <i>(недопустимы для неупорядоченных ассоциативных контейнеров)</i>
Получения итераторов	
<code>c.begin(), c.end()</code>	Возвращают итератор на первый и следующий после последнего элемент в контейнере <code>c</code>
<code>c.cbegin(), c.cend()</code>	Возвращают <code>const_iterator</code>
Дополнительные члены реверсивных контейнеров (недопустимы для <code>forward_list</code>)	
<code>reverse_iterator</code>	Итератор, обеспечивающий доступ к элементам в обратном порядке
<code>const_reverse_iterator</code>	Реверсивный итератор, не позволяющий запись в элементы
	Возвращает итератор на последний и следующий после

c. rbegin(), c. rend()	первого элементы контейнера с
c. crbegin(), c. crend()	Возвращают итератор const_reverse_iterator

В этом разделе рассматриваются аспекты, являющиеся общими для всех контейнеров. Остальная часть этой главы посвящена исключительно последовательным контейнерам; операции, специфические для ассоциативных контейнеров, рассматриваются в главе 11.

Обычно каждый контейнер определяется в файле заголовка, название которого совпадает с именем типа. Таким образом, тип `deque` определен в заголовке `deque`, тип `list` — в заголовке `list` и т.д. Контейнеры — это шаблоны классов (см. раздел 3.3). Подобно векторам, при создании контейнера специфического типа необходимо предоставить дополнительную информацию. Для большинства контейнеров, но не всех, предоставляемой информацией является тип элемента:

```
list<Sales_data> // список, содержащий объекты
класса Sales_data
deque<double> // двухсторонняя очередь
переменных типа double
```

Ограничения на типы, которые может содержать контейнер

Типом элемента последовательного контейнера может быть практически любой тип. В частности, типом элемента контейнера может быть другой контейнер. Такие контейнеры определяют точно так же, как любые другие: в угловых скобках указывается тип элемента (которым в данном случае является другой контейнер):

```
vector<vector<string>> lines; // вектор векторов
где lines — это вектор, элементами которого являются векторы
строк.
```



Устаревшие компиляторы могут потребовать пробела между угловыми скобками, например `vector<vector<string> >`.

Несмотря на то что в контейнере можно хранить практически любой тип, некоторые операции налагают на тип элемента собственные требования. Можно определить контейнер для типа, который не

поддерживает определенное операцией требование, но использовать операцию можно, только если тип элемента отвечает требованиям этой операции.

В качестве примера рассмотрим конструктор последовательного контейнера, получающий аргумент размера (см. раздел 3.3.1) и использующий стандартный конструктор типа элемента. У некоторых классов нет стандартного конструктора. Вполне можно определить контейнер, содержащий объекты такого типа, но создать такой контейнер, используя только количество элементов, нельзя:

```
// тип noDefault не имеет стандартного конструктора
vector<noDefault> v1(10, init); // ok: предоставлен
инициализатор
                                            // элемента
vector<noDefault> v2(10);                // ошибка:
необходимо предоставить
                                            // инициализатор
элемента
```

Поскольку рассматриваются контейнерные операции, следует заметить, что тип элемента накладывает дополнительные ограничения, если таковые вообще имеются, на каждую операцию с контейнером.

Упражнения раздела 9.2

Упражнение 9.2. Определите список (`list`), элементами которого будут двухсторонние очереди целых чисел.



9.2.1. Итераторы

Подобно контейнерам, у итераторов есть общий интерфейс: если итератор поддерживает некую функцию, то аналогичным образом она работает с каждым поддерживающим ее итератором. Например, итераторы всех контейнеров стандартных типов позволяют обращаться к элементу, предоставляя оператор обращения к значению. Аналогично все итераторы библиотечных контейнеров определяют оператор инкремента для перемещения от одного элемента к следующему.

За одним исключением контейнерные итераторы поддерживают все функции, перечисленные в табл. 3.6. Исключение в том, что итераторы

контейнера `forward_list` не поддерживают оператор декремента (`--`). Операторы арифметических действий с итераторами, перечисленными в табл. 3.7, применимы только к итераторам контейнеров `string`, `vector`, `deque` и `array`. К итераторам контейнеров любых других типов эти операторы неприменимы.

Диапазоны итераторов



Концепция диапазона итераторов фундаментальна для стандартной библиотеки.

Диапазон итераторов (iterator range) обозначается парой итераторов, каждый из которых указывает на элемент или на *следующий элемент после последнего* в том же контейнере. Эти два итератора, обозначающие диапазон элементов контейнера, зачастую называют `begin` и `end` или, что несколько обманчиво, `first` и `last`.

Хоть имя `last` и общепринято, оно немного вводит в заблуждение, поскольку второй итератор никогда не указывает на последний элемент диапазона. Вместо этого он указывает на позицию следующего элемента после последнего. Диапазон включает элемент, обозначенный итератором `first`, и все элементы от него до обозначенного итератором `last`, но не включая его.

Такой диапазон элементов называется *интервал, включающий левый элемент* (left-inclusive interval). Вот стандартная математическая форма записи такого диапазона:

[`begin`, `end`)

Это указывает, что диапазон начинается с элемента, обозначенного итератором `begin`, и заканчивается элементом перед тем, который обозначен итератором `end`. Итераторы `begin` и `end` должны относиться к тому же контейнеру. Итератор `end` может быть равен итератору `begin`, но не должен указывать на элемент перед обозначенным итератором `begin`.

Требования к итераторам, формирующими диапазон

Два итератора, `begin` и `end`, позволяют задать диапазон при следующих условиях.

- Итераторы относятся к существующим элементам или к

следующему элементу за концом того же контейнера.

- Элемент `end` достичим благодаря последовательному приращению итератора `begin`. Другими словами, итератор `end` не должен предшествовать итератору `begin`.



Компилятор не может сам соблюдать эти требования. Позаботиться об этом придется разработчику.

Смысл использования диапазонов, включающих левый элемент

Библиотека использует диапазоны, включающие левый элемент, потому, что они обладают двумя очень полезными качествами (напомним, что допустимый диапазон обозначают итераторы `begin` и `end`).

- Если итератор `begin` равен итератору `end`, то диапазон пуст.
- Если итератор `begin` не равен итератору `end`, в диапазоне содержится по крайней мере один элемент и итератор `begin` указывает на первый из них.
- Можно осуществлять инкремент итератора `begin` до тех пор, пока он не станет равен итератору `end` (т.е. `begin == end`).

Благодаря этим качествам можно создавать вполне безопасные циклы обработки диапазона элементов, например, такие:

```
while (begin != end) {  
    *begin = val; // ok: диапазон не пуст, begin  
обозначает элемент  
    ++begin;       // переместить итератор и получить  
следующий элемент  
}
```

Если итераторы `begin` и `end` задают допустимый диапазон элементов, выполнение условия `begin == end` означает, что диапазон пуст. В данном случае это условие выхода из цикла. Если диапазон не пуст, значит, итератор `begin` указывает на элемент в этом не пустом диапазоне. Вполне очевидно, что в теле цикла `while` можно безопасно обращаться к значению итератора `begin`, поскольку оно гарантировано существует. И наконец, поскольку инкремент итератора `begin` осуществляется в теле цикла, последний гарантированно будет конечным.

Упражнения раздела 9.2.1

Упражнение 9.3. Каким условиям должны удовлетворять итераторы, обозначающие диапазон?

Упражнение 9.4. Напишите функцию, которая получает два итератора вектора `vector<int>` и значение типа `int`. Организуйте поиск этого значения в диапазоне и возвратите логическое значение (тип `bool`), указывающее, что значение найдено.

Упражнение 9.5. Перепишите предыдущую программу так, чтобы она возвращала итератор на найденный элемент. Функция должна учитывать случай, когда элемент не найден.

Упражнение 9.6. Что не так со следующей программой? Как ее можно исправить?

```
list<int> lst1;
list<int>::iterator iter1 = lst1.begin(),
iter2 = lst1.end();
while (iter1 < iter2) /* ... */
```

9.2.2. Типы-члены классов контейнеров

Класс каждого контейнера определяет несколько типов, представленных в табл. 9.2. Три из них уже использовались: `size_type` (см. раздел 3.2.2), `iterator` и `const_iterator` (см. раздел 3.4.1).

Кроме итераторов уже использовавшихся типов, большинство контейнеров предоставляет *реверсивные итераторы* (`reverse_iterator`). Другими словами, реверсивный итератор — это итератор, перебирающий контейнер назад и инвертирующий значение его операторов. Например, оператор `++` возвращает реверсивный итератор к предыдущему элементу. Более подробная информация о реверсивных итераторах приведена в разделе 10.4.3.

Остальные псевдонимы типов позволяют использовать тип хранящихся в контейнере элементов, даже не зная его конкретно. Если необходим тип элемента, используется тип `value_type` контейнера. Если необходима ссылка на этот тип, используется член `reference` или `const_reference`. Эти связанные с элементами псевдонимы типов весьма полезны в обобщенном программировании, которое рассматривается в главе 16.

Чтобы использовать один из этих типов, следует указать класс, членами которого они являются

```
// iter имеет тип iterator, определенный классом
list<string>
```

```

list<string>::iterator iter;
// count имеет тип difference_type, определенный
классом vector<int>
vector<int>::difference_type count;

```

В этих объявлениях оператор области видимости (см. раздел 1.2) позволяет указать, что используется тип-член iterator класса list<string> и тип-член difference_type, определенный классом vector<int>, соответственно.

Упражнения раздела 9.2.2

Упражнение 9.7. Какой тип следует использовать в качестве индекса для вектора целых чисел?

Упражнение 9.8. Какой тип следует использовать для чтения элементов в списке строк?



9.2.3. Функции-члены begin() и end()

Функции-члены begin() и end() (см. раздел 3.4.1) возвращают итераторы на первый и следующий после последнего элементы контейнера соответственно. Эти итераторы, как правило, используют при создании диапазона итераторов, охватывающего все элементы контейнера.

Как показано в табл. 9.2, есть несколько версий этих функций: имена которых начинаются с буквы r возвращают реверсивные итераторы (рассматриваются в разделе 10.4.3), а с буквы c — возвращают константную версию соответствующего итератора:

```

list<string>    a      = {"Milton",     "Shakespeare",
"Austen"} ;
auto it1 = a.begin(); // list<string>::iterator
auto      it2      =      a.rbegin(); // 
list<string>::reverse_iterator
auto      it3      =      a.cbegin(); // 
list<string>::const_iterator
auto      it4      =      a.crbegin(); // 
list<string>::const_reverse_iterator

```

Функции, имена которых не начинаются с буквы c, перегружены. Таким образом, фактически есть две функции-члена begin(). Одна

является константной (см. раздел 7.1.2) и возвращает тип `const_iterator` контейнера. Вторая не константна и возвращает тип `iterator` контейнера. Аналогично для функций `rbegin()`, `end()` и `rend()`. При вызове такой функции-члена для неконстантного объекта используется версия, возвращающая тип `iterator`. Константная версия итераторов будет получена *только* при вызове этих функций для константного объекта. Подобно указателям и ссылкам на константу, итератор типа `iterator` можно преобразовать в соответствующий итератор типа `const_iterator`, но не наоборот.



Версии этих функций, имена которых не начинаются с буквы `c`, были введены согласно новому стандарту для обеспечения использования ключевого слова `auto` с функциями `begin()` и `end()` (см. раздел 2.5.2). Прежде не было никакого иного выхода, кроме как явно указать необходимый тип итератора:

```
// тип указан явно
list<string>::iterator it5 = a.begin();
list<string>::const_iterator it6 = a.begin();
// iterator или const_iterator в зависимости от
типа a
auto it7 = a.begin(); // const_iterator только
если a константа
auto it8 = a.cbegin(); // it8 - const_iterator
```

Когда с функциями `begin()` или `end()` используется ключевое слово `auto`, тип возвращаемого итератора зависит от типа контейнера. То, как предполагается использовать итератор, несущественно. Версии с позволяют получать итератор типа `const_iterator` независимо от типа контейнера.

Рекомендуем

Когда доступ на запись не нужен, используйте версии `cbegin()` и `cend()`.

Упражнения раздела 9.2.3

Упражнение 9.9. В чем разница между функциями `begin()` и

`cbegin()`?

Упражнение 9.10. Каковы типы следующих четырех объектов?

```
vector<int> v1;  
const vector<int> v2;  
auto it1 = v1.begin(), it2 = v2.begin();  
auto it3 = v1.cbegin(), it4 = v2.cbegin();
```



9.2.4. Определение и инициализация контейнера

Каждый контейнерный тип определяет стандартный конструктор (см. раздел 7.1.4). За исключением контейнера `array` стандартный конструктор создает пустой контейнер определенного типа. Также за исключением контейнера `array` другие конструкторы получают аргументы, которые определяют размер контейнера и исходные значения его элементов.

Инициализация контейнера как копии другого контейнера

Существуют два способа создать новый контейнер как копию другого: можно непосредственно скопировать контейнер или (за исключением контейнера `array`) скопировать только диапазон его элементов, обозначенный парой итераторов.

Таблица 9.3. Определение и инициализация контейнера

<code>C c;</code>	Стандартный конструктор. Если <code>C</code> — массив, то элементы контейнера <code>c</code> инициализируются значением по умолчанию; в противном случае контейнер <code>c</code> пуст
<code>C c1(c2) C c1 = c2</code>	Контейнер <code>c1</code> — копия <code>c2</code> . Контейнеры <code>c1</code> и <code>c2</code> должны быть одинакового типа (т.е. должны иметь тот же тип контейнера и содержать элементы того же типа; у массивов должен также совпадать размер)
<code>C c{ a, b, c... }</code> <code>C c = { a, b, c... }</code>	Контейнер <code>c</code> содержит копию элементов из списка инициализации. Тип элементов в списке должен быть совместимым с типом элементов <code>C</code> . В случае массива количество элементов списка не должно превышать размер массива, а все недостающие элементы инициализируются значением по умолчанию (см. раздел 3.3.1)
	Контейнер <code>c</code> содержит копию элементов из диапазона, обозначенного итераторами <code>b</code> и <code>e</code> . Тип элементов должен быть

C c(b, e)	совместимым с типом элементов C. (Недопустимо для массива.)
Получающие размер конструкторы допустимы только для последовательных контейнеров (исключая массив)	
C seq(n)	Контейнер seq содержит n элементов, инициализированных значением по умолчанию; этот конструктор является явным (см. раздел 7.5.4). (Недопустимо для строки.)
C seq(n, t)	Контейнер seq содержит n элементов со значением t

Чтобы создать контейнер как копию другого, их типы контейнеров и элементов должны совпадать. При передаче итераторов идентичность типов контейнеров необязательна. Кроме того, могут отличаться типы элементов нового и исходного контейнеров, если возможно преобразование между ними (см. раздел 4.11):

```
// каждый контейнер имеет три элемента,
инициализированных
// предоставленными инициализаторами
list<string> authors = {"Milton", "Shakespeare",
"Austen"} ;
vector<const char*> articles = {"a", "an", "the"} ;
list<string> list2( authors); // ok: типы
совпадают
deque<string> authList( authors); // ошибка: типы
контейнеров
// не совпадают
vector<string> words( articles); // ошибка: типы
элементов не совпадают
// ok: преобразует элементы const char* в string
forward_list<string> words( articles.begin(),
articles.end());
```



При копировании содержимого одного контейнера в другой типы контейнеров и их элементов должны точно совпадать.

Конструктор, получающий два итератора, использует их для обозначения диапазона копируемых элементов. Как обычно, итераторы

отмечают первый и следующий элемент после последнего копируемого. Размер нового контейнера будет совпадать с количеством элементов в диапазоне. Каждый элемент в новом контейнере инициализируется значением соответствующего элемента в диапазоне.

Поскольку итераторы обозначают диапазон, этот конструктор можно использовать для копирования части последовательности. С учетом того что `it` является итератором, обозначающим элемент в контейнере `authors`, можно написать следующее:

```
// копирует до, но не включая элемент, обозначенный  
итератором it  
deque<string> authList(authors.begin(), it);
```

Списочная инициализация



По новому стандарту контейнер допускает списочную инициализацию (см. раздел 3.3.1):

```
// каждый контейнер имеет три элемента,  
инициализированных  
// предоставленными инициализаторами  
list<string> authors = {"Milton", "Shakespeare",  
"Austen"};  
vector<const char*> articles = {"a", "an", "the"};
```

Это определяет значение каждого элемента в контейнере явно. Кроме контейнеров таких типов, как `array`, список инициализации неявно определяет также размер контейнера: у контейнера будет столько элементов, сколько инициализаторов в списке.

Конструкторы последовательных контейнеров, связанные с размером

Кроме конструкторов, общих для последовательных и ассоциативных контейнеров, последовательные контейнера (кроме массива) можно также инициализировать, указав их размера и (необязательного) инициализирующий элемент. Если его не предоставить, библиотека создает инициализирующее значение сама (см. раздел 3.3.1):

```
vector<int> ivec(10, -1); // десять элементов  
типа int; значение -1  
list<string> svec(10, "hi!"); // десять строк;
```

```
значение "hi!"  
forward_list<int> ivect(10); // десять элементов;  
значение 0  
deque<string> svec(10); // десять элементов;  
все пустые строки
```

Конструктор, получающий аргумент размера, можно также использовать, если элемент имеет встроенный тип или тип класса, у которого есть стандартный конструктор (см. раздел 9.2). Если у типа элемента нет стандартного конструктора, то наряду с размером следует определить явный инициализатор элемента.



Конструкторы, получающие размер, допустимы *только* для последовательных контейнеров; ассоциативные контейнеры их не поддерживают.

Библиотечные массивы имеют фиксированный размер

Подобно тому, как размер встроенного массива является частью его типа, размер библиотечного контейнера `array` тоже входит в состав его типа. Когда определяется массив, кроме типа элемента задается также размер контейнера:

```
array<int, 42> // тип: массив, содержащий 42  
целых числа  
array<string, 10> // тип: массив, содержащий 10  
строк
```

Чтобы использовать контейнер `array`, следует указать тип элемента и его размер:

```
array<int, 10>::size_type i; // тип массива  
включает тип элемента  
// и размер  
array<int>::size_type j; // ошибка: array<int>  
- это не тип
```

Поскольку размер является частью типа массива, контейнер `array` не поддерживает обычные конструкторы контейнерных классов. Эти конструкторы, явно или неявно, определяют размер контейнера. В случае массива разрешение пользователям передавать аргумент размера было бы

избыточно (в лучшем случае) и приводило бы к ошибкам.

Фиксированный характер размера массивов влияет также на поведение остальных конструкторов, действительно определяющих массив. В отличие от других контейнеров, созданный по умолчанию массив не пуст: количество его элементов соответствует размеру, а инициализированы они значением по умолчанию (см. раздел 2.2.1), как и элементы встроенного массива (см. раздел 3.5.1). При списочной инициализации массива количество инициализаторов не должно превышать размер массива. Если инициализаторов меньше, чем элементов массива, они используются для первых элементов, а все остальные инициализируются значением по умолчанию (см. раздел 3.3.1). В любом случае, если типом элемента является класс, то у него должен быть стандартный конструктор, обеспечивающий инициализацию значением по умолчанию:

```
array<int, 10> ia1;           // десять целых чисел,
инициализированных
                                         // значением по
умолчанию
array<int, 10> ia2 = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; // /
списочная инициализация
array<int, 10> ia3 = { 42 }; // ia3[0] содержит
значение 42, остальные
                                         // элементы - значение 0
```

Следует заметить, что хоть и нельзя копировать или присваивать объекты массивов встроенных типов (см. раздел 3.5.1), для контейнеров `array` такого ограничения нет:

```
int digs[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int cpy[10] = digs; // ошибка: встроенные массивы
не допускают
                                         // копирования и присвоения
array<int, 10> digits = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
array<int, 10> copy = digits; // ok: если типы
массивов совпадают
```

Как обычно, инициализирующий контейнер должен иметь тот же тип, что и создаваемый. Для контейнера `array` совпадать должны тип элемента и размер, поскольку размер массива — часть его типа.

Упражнения раздела 9.2.4

Упражнение 9.11. Приведите пример каждого из шести способов создания и инициализации контейнеров `vector`. Объясните, какие

значения будет содержать каждый вектор.

Упражнение 9.12. Объясните различие между конструктором, получающим контейнер для копирования, и конструктором получающим два итератора.

Упражнение 9.13. Как инициализировать контейнер `vector<double>` из контейнера `list<int>` и контейнера `vector<int>`? Напишите код для проверки ответов.

9.2.5. Присвоение и функция `swap()`

Связанные с присвоением операторы, перечисленные в табл. 9.4, действуют на весь контейнер. Оператор присвоения заменяет весь диапазон элементов в левом контейнере копиями элементов из правого:

```
c1 = c2;           // заменяет содержимое контейнера c1
копией
                           // элементов контейнера c2
c1 = { a, b, c}; // после присвоения контейнер c1
имеет размер 3
```

После первого присвоения контейнеры слева и справа равны. Если контейнеры имели неравный размер, после присвоения у обоих будет размер контейнера из правого операнда. После второго присвоения размер контейнера `c1` составит 3, что соответствует количеству значений, представленных в списке.

Таблица 9.4. Операторы присвоения контейнеров

<code>c1 = c2</code>	Заменяет элементы контейнера <code>c1</code> копиями элементов контейнера <code>c2</code> . Контейнеры <code>c1</code> и <code>c2</code> должны иметь тот же тип
<code>c = { a, b,</code> <code>c... }</code>	Заменяет элементы контейнера <code>c1</code> копиями элементов списка инициализации. (Недопустимо для массива.)
<code>swap(c1, c2)</code> <code>c1.swap(c2)</code>	Обменивает элементы контейнеров <code>c1</code> и <code>c2</code> . Контейнеры <code>c1</code> и <code>c2</code> должны иметь тот же тип. Обычно функция <code>swap()</code> выполняется намного быстрее, чем процесс копирования элементов из контейнера <code>c2</code> в <code>c1</code>
Операторы присвоения недопустимы для ассоциативных контейнеров и массива	
<code>seq.assign(b, e)</code>	Заменяет элементы в контейнере <code>seq</code> таковыми из диапазона, обозначенного итераторами <code>b</code> и <code>e</code> . Итераторы <code>b</code> и <code>e</code> не должны ссылаться на элементы в контейнере <code>seq</code>
<code>seq.assign(il)</code>	Заменяет элементы в контейнере <code>seq</code> таковыми из списка инициализации <code>il</code>

seq.assign(n, t)	Заменяет элементы в контейнере seq набором из n элементов со значением t
--------------------	--

В отличие от встроенных массивов, библиотечный тип array поддерживает присвоение. У левых и правых операндов должен быть одинаковый тип:

```
array<int, 10> a1 = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;
array<int, 10> a2 = { 0 }; // все элементы со
значением 0
a1 = a2; // замена элементов в a1
a2 = { 0 }; // ошибка: нельзя присвоить массиву
значения из списка
```

Поскольку размер правого операнда может отличаться от размера левого операнда, тип array не поддерживает функцию assign() и это не позволяет присваивать значения из списка.



Связанные с присвоением операторы делают недопустимыми итераторы, ссылки и указатели на контейнер слева.

Применение функции assign() (только последовательные контейнеры)

Оператор присвоения требует совпадения типов левых и правых операндов. Он копирует все элементы правого операнда в левый. Последовательные контейнеры (кроме array) определяют также функцию-член assign(), обеспечивающую присвоение разных, но совместимых типов, или присвоение контейнерам последовательностей. Функция assign() заменяет все элементы в левом контейнере копиями элементов, указанных в ее аргументе. Например, функцию assign() можно использовать для присвоения диапазона значений char* из вектора в список строк:

```
list<string> names;
vector<const char*> oldstyle;
names = oldstyle; // ошибка: типы контейнеров не
совпадают
// ok: преобразование из const char* в string
возможно
```

```
names.assign( oldstyle.cbegin(), oldstyle.cend() );
```

Вызов функции `assign()` заменяет элементы в контейнере `names` копиями элементов из диапазона, обозначенного итераторами. Аргументы функции `assign()` определяют количество элементов контейнера и их значения.



Поскольку существующие элементы заменяются, итераторы, переданные функции `assign()`, не должны относиться к контейнеру, для которого вызвана функция `assign()`.

Вторая версия функции `assign()` получает целочисленное значение и значение элемента. Она заменяет указанное количество элементов контейнера заданным значением:

```
// эквивалент slist1.clear();
// сопровождается slist1.insert(slist1.begin(), 10,
"Hiya!");
list<string> slist1(1);           // один элемент; пустая
строка
slist1.assign(10, "Hiya!"); // десять элементов; со
значением "Hiya!"
```

Применение функции `swap()`

Функция `swap()` меняет местами значения двух контейнеров того же типа. Типы контейнеров и их элементов должны совпадать. После обращения к функции `swap()` элементы правого операнда оказываются в левом, и наоборот:

```
vector<string> svec1(10); // вектор из 10 элементов
vector<string> svec2(24); // вектор из 24 элементов
svec1.swap(svec2);
```

После выполнения функции `swap()` вектор `svec1` содержит 24 строки, а вектор `svec2` — 10. За исключением массивов смена двух контейнеров осуществляется очень быстро — сами элементы не меняются; меняются лишь внутренние структуры данных.



За исключением массивов функция `swap()` не копирует, не удаляет и не вставляет элементы, поэтому она гарантированно выполняется за постоянное время.

Благодаря тому факту, что элементы не перемещаются, итераторы, ссылки и указатели в контейнере, за исключением контейнера `string`, остаются допустимыми. Они продолжают указывать на те же элементы, что и перед перестановкой. Однако после вызова функции `swap()` эти элементы находятся в другом контейнере. Предположим, например, что итератор `iter` обозначал в векторе строк позицию `svec1[3]`. После вызова функции `swap()` он обозначит элемент в позиции `svec2[3]`. В отличие от других контейнеров, вызов функции `swap()` для строки делает некорректными итераторы, ссылки и указатели.

В отличие от поведения функции `swap()` с другими контейнерами, когда дело доходит до массивов, элементы действительно обмениваются. Обмен двух массивов потребует времени, пропорционального количеству их элементов.

После выполнения функции `swap()`, указатели, ссылки и итераторы остаются связанными с тем же элементом, который они обозначили ранее. Конечно, значение самого элемента заменено значением соответствующего элемента другого массива.



Новая библиотека предоставляет функцию `swap()` в версии члена класса контейнера и в версии, не являющейся членом какого-либо класса. Прежние версии библиотеки определили только версию функции-члена `swap()`. Функция `swap()` не являющаяся членом класса контейнера, имеет большое значение в обобщенных программах. Как правило, лучше использовать именно эту версию.

Упражнения раздела 9.2.5

Упражнение 9.14. Напишите программу, присваивающую значения элементов списка указателей на символьные строки в стиле С (тип `char*`) элементам вектора строк.



9.2.6. Операции с размером контейнера

За одним исключением у классов контейнеров есть три функции, связанные с размером. Функция-член `size()` (см. раздел 3.2.2) возвращает количество элементов в контейнере; функция-член `empty()` возвращает логическое значение `true`, если контейнер пуст, и значение `false` в противном случае; функция-член `max_size()` возвращает число, большее или равное количеству элементов, которые может содержать контейнер данного типа. По причинам, рассматриваемым в следующем разделе, контейнер `forward_list` предоставляет функции `max_size()` и `empty()`, но не функцию `size()`.

9.2.7. Операторы сравнения

Для сравнения используется тот же реляционный оператор, который определен для типа элементов: при сравнении двух контейнеров на неравенство (`!=`) используется оператор `!=` типа их элементов. Если тип элемента не поддерживает определенный оператор, то для сравнения контейнеров такого типа данный оператор использовать нельзя.

Сравнение двух контейнеров осуществляется на основании сравнения пар их элементов. Эти операторы работают так же, как и таковые у класса `string` (см. раздел 3.2.2):

- Если оба контейнера имеют одинаковый размер и все их элементы совпадают, контейнеры равны, в противном случае — не равны.
- Если контейнеры имеют различный размер, но каждый элемент короткого совпадает с соответствующим элементом длинного, считается, что короткий контейнер меньше длинного.
- Если значения элементов контейнеров не совпадают, результат их сравнения зависит от значений первых неравных элементов.

Проще всего понять работу операторов, рассмотрев их на примерах.

```
vector<int> v1 = { 1, 3, 5, 7, 9, 12 };  
vector<int> v2 = { 1, 3, 9 };  
vector<int> v3 = { 1, 3, 5, 7 };  
vector<int> v4 = { 1, 3, 5, 7, 9, 12 };  
v1 < v2 // true; v1 и v2 отличаются элементом [2]:  
v1[2] меньше,
```

```

    // чем v2[ 2 ]
v1 < v3 // false; все элементы равны, но у v3 их
меньше;
v1 == v4 // true; все элементы равны и размер v1 и
v4 одинаков
v1 == v2 // false; v2 имеет меньше элементов, чем
v1

```

При сравнении контейнеров используются операторы сравнения их элементов



Сравнить два контейнера можно только тогда, когда используемый оператор сравнения определен для типа элемента контейнера.

Операторы равенства контейнеров используют оператор == элемента, а операторы сравнения — оператор < элемента. Если тип элемента не предоставляет необходимый оператор, то не получится использовать соответствующие операторы и с содержащими их контейнерами. Например, определенный в главе 7 тип Sales_data не предоставлял операторов == и <. Поэтому нельзя сравнить два контейнера, содержащих элементы типа Sales_data:

```

vector<Sales_data> storeA, storeB;
if (storeA < storeB) // ошибка: Sales_data не имеет
оператора <

```

Упражнения раздела 9.2.7

Упражнение 9.15. Напишите программу, выясняющую, равны ли два вектора `vector<int>`.

Упражнение 9.16. Перепишите предыдущую программу, но сравните элементы списка `list<int>` и вектора `vector<int>`.

Упражнение 9.17. Допустим, `c1` и `c2` являются контейнерами. Какие условия налагают типы их элементов в следующем выражении?

```
if (c1 < c2)
```

9.3. Операции с последовательными контейнерами

Последовательные и ассоциативные контейнеры отличаются организацией своих элементов. Это различие влияет на способ хранения, доступа, добавления и удаления элементов. В предыдущем разделе рассматривались операции, общие для всех контейнеров (см. табл. 9.2). В остальной части этой главы рассматриваются операции, специфические для последовательных контейнеров.



9.3.1. Добавление элементов в последовательный контейнер

За исключением массива все библиотечные контейнеры обеспечивают гибкое управление памятью. Они позволяют добавлять и удалять элементы, динамически изменяя размер контейнера во время выполнения. В табл. 9.5 перечислены функции, позволяющие добавлять элементы в последовательный контейнер (но не в массив).

Используя эти функции, следует помнить, что контейнеры применяют различные стратегии резервирования памяти для элементов и что эти стратегии влияют на производительность. Например, добавление элементов в любое место вектора или строки (но не в конец) либо в любое место двухсторонней очереди (но не в начало или в конец) требует перемещения элементов.

Кроме того, добавление элементов в вектор или строку может привести к повторному резервированию памяти всего объекта. Это, в свою очередь, потребует резервирования новой памяти для элементов и их перемещения из старых областей в новые.

Таблица 9.5. Функции, добавляющие элементы в последовательный контейнер

Эти функции изменяют размер контейнера; они не поддерживаются массивами. Контейнер `forward_list` обладает специальными версиями функций `insert()` и `emplace()`; см. раздел 9.3.4, а функций `push_back()` и `emplace_back()` у него нет. Функции `push_front()` и `emplace_front()` недопустимы для контейнеров `vector` и `string`.

c. <code>push_back(t)</code> c. <code>emplace_back(args)</code>	Создает в конце контейнера с элементом со значением <code>t</code> или переданным аргументом <code>args</code> . Возвращает тип <code>void</code>
c. <code>push_front(t)</code> c. <code>emplace_front(args)</code>	Создает в начале контейнера с элементом со значением <code>t</code> или переданным аргументом <code>args</code> . Возвращает тип <code>void</code>
c. <code>insert(p, t)</code> c. <code>emplace(p, args)</code>	Создает элемент со значением <code>t</code> или переданным аргументом <code>args</code> перед элементом, обозначенным итератором <code>p</code> . Возвращает итератор на добавленный элемент
	Вставляет <code>n</code> элементов со значением <code>t</code> перед элементом,

c.insert(p, n, t)	обозначенным итератором p. Возвращает итератор на первый вставленный элемент; если n — нуль, возвращается итератор p
c.insert(p, b, e)	Вставляет элементы из диапазона, обозначенного итераторами b и e перед элементом, обозначенным итератором p. Итераторы b и e не могут относиться к элементам в контейнере c. Возвращает итератор на первый вставленный элемент; если диапазон пуст, возвращается итератор p
c.insert(p, il)	il — это список значений элементов. Вставляет переданные значения перед элементом, обозначенным итератором p. Возвращает итератор на первый вставленный элемент; если список пуст, возвращается итератор p

*Применение функции **push_back()***

В разделе 3.3.2 упоминалось, что функция `push_back()` добавляет элемент в конец вектора. Кроме контейнеров `array` и `forward_list`, каждый последовательный контейнер (включая `string`) поддерживает функцию `push_back()`.

Цикл следующего примера читает по одной строке за раз в переменную `word`:

```
// читать слова со стандартного устройства ввода и
помещать
// их в конец контейнера
string word;
while (cin >> word)
    container.push_back( word );
```

Вызов функции `push_back()` создает новый элемент в конце контейнера `container`, увеличивая его размер на 1. Значением этого элемента будет копия значения переменной `word`. Контейнер может иметь любой тип: `list`, `vector` или `deque`.

Поскольку класс `string` — это только контейнер символов, функцию `push_back()` можно использовать для добавления символов в ее конец:

```
void pluralize( size_t cnt, string &word) {
    if (cnt > 1)
        word.push_back('s'); // то же, что и word += 's'
}
```

Ключевая концепция. Элементы контейнера содержат копии значений

Когда объект используется для инициализации контейнера или вставки в него, в контейнер помещается копия значения объекта, а не сам объект. Подобно передаче объекта в не ссылочном параметре (см. раздел 6.2.1), между элементом в контейнере и объектом, значение которого было использовано для его инициализации, нет никакой связи. Последующие изменения элемента в контейнере никак не влияют на исходный объект, и наоборот.

Применение функции `push_front()`

Кроме функции `push_back()`, контейнеры `list`, `forward_list` и `deque` предоставляют аналогичную функцию `push_front()`. Она вставляет новый элемент в начало контейнера:

```
list<int> ilist;
// добавить элементы в начало ilist
for (size_t ix = 0; ix != 4; ++ix)
    ilist.push_front(ix);
```

Этот цикл добавляет элементы 0, 1, 2, 3 в начало списка `ilist`. Каждый элемент вставляется в *новое начало* списка, т.е. когда добавляется 1, она оказывается перед 0, 2 — перед 1 и так далее. Таким образом, добавленные в цикле элементы расположены в обратном порядке. После этого цикла список `ilist` содержит такую последовательность: 3, 2, 1, 0.

Обратите внимание: контейнер `deque`, предоставляющий подобно контейнеру `vector` быстрый произвольный доступ к своим элементам, обладает функцией-членом `push_front()`, а контейнер `vector` — нет. Контейнер `deque` гарантирует постоянное время вставки и удаления элементов в начало и в конец контейнера. Как и у контейнера `vector`, вставка элементов иначе как в начало или в конец контейнера `deque` — потенциально продолжительная операция.



Добавление элементов в контейнеры `vector`, `string` или `deque` способно сделать недействительными все существующие итераторы, ссылки и указатели на их элементы.

Добавление элементов в указанную точку контейнера

Функции `push_back()` и `push_front()` предоставляют весьма

удобный способ добавления одиночных элементов в конец или в начало последовательного контейнера. Функция `insert()` обладает более общим характером и позволяет вставлять любое количество элементов в любую указанную позицию контейнера. Ее поддерживают контейнеры `vector`, `deque`, `list` и `string`, а контейнер `forward_list` предоставляет собственные специализированные версии этих функций-членов, которые рассматриваются в разделе 9.3.4.

Каждая из функций вставки получает в качестве первого аргумента итератор, указывающий позицию помещения элемента (элементов) в контейнере. Он может указывать на любую позицию, включая следующий элемент после конца контейнера. Поскольку итератор может указывать на несуществующий элемент после конца контейнера, а также потому, что полезно иметь способ вставки элементов в начало контейнера, элемент (элементы) вставляются *перед* позицией, обозначенной итератором. Рассмотрим следующий оператор:

```
slist.insert( iter, "Hello! " ); // вставить "Hello!"  
прямо перед iter
```

Он вставляет строку со значением "Hello" непосредственно перед элементом, обозначенным итератором `iter`.

Даже при том, что у некоторых контейнеров нет функции `push_front()`, к функции `insert()` это не относится. Функция `insert()` позволяет вставлять элементы в начало контейнера, не заботясь о наличии у контейнера функции `push_front()`:

```
vector<string> svec;  
list<string> slist;  
// эквивалент вызова slist.push_front("Hello!");  
slist.insert(slist.begin(), "Hello!");  
// вектор не имеет функции push_front(),  
// но вставка перед begin() возможна  
// внимание: вставка возможна везде, но вставка в  
конец вектора может  
// потребовать больше времени  
svec.insert(svec.begin(), "Hello!");
```



ВНИМАНИЕ

Контейнеры `vector`, `deque` и `string` допускают вставку в любую позицию, но это может потребовать больше времени.

Вставка диапазона элементов

Следующие аргументы функции `insert()`, расположенные после начального итератора, похожи на аргументы конструкторов контейнеров. Версия, получающая количество элементов и значение, добавляет определенное количество одинаковых элементов перед указанной позицией:

```
svec.insert(svec.end(), 10, "Anna");
```

Этот код вставляет 10 элементов в конец вектора `svec` и инициализирует каждый из них строкой "Anna".

Данная версия функции `insert()` получает пару итераторов или список инициализации для вставки элементов из данного диапазона перед указанной позицией:

```
vector<string> v = {"quasi", "simba", "frollo",
"scar"};
// вставить два последних элемента вектора v в
начало slist
slist.insert(slist.begin(), v.end() - 2, v.end());
slist.insert(slist.end(), {"these", "words",
"will",
"go", "at", "the",
"end"});
// ошибка времени выполнения:
// обозначающие копируемый диапазон итераторы
// не должны принадлежать тому же контейнеру,
// который изменяется
slist.insert(slist.begin(), slist.begin(),
slist.end());
```

При передаче пары итераторов они не могут относиться к тому же контейнеру, к которому добавляются элементы.



По новому стандарту версии функции `insert()`, получающие количество или диапазон, возвращают итератор на первый вставленный элемент. (В предыдущих версиях библиотеки эти функции возвращали тип `void`.) Если диапазон пуст, никакие элементы не вставляются, а функция возвращает свой первый параметр.

Применение возвращаемого значения функции `insert()`

Значение, возвращенное функцией `insert()`, можно использовать для многократной вставки элементов в определенной позиции контейнера:

```
list<string> lst;
auto iter = lst.begin();
while (cin >> word)
    iter = lst.insert(iter, word); // то же, что и
вызов push_front()
```



Важно понимать, почему именно цикл, подобный этому, эквивалентен вызову функции `push_front()`.

Перед циклом итератор `iter` инициализируется возвращаемым значением функции `lst.begin()`. Первый вызов функции `insert()` получает только что прочитанную строку и помещает ее перед элементом, обозначенным итератором `iter`. Значение, возвращенное функцией `insert()`, является итератором на этот новый элемент. Присвоим этот итератор итератору `iter` и повторим цикл, читая другое слово. Пока есть слова для вставки, каждая итерация цикла `while` вставляет новый элемент перед позицией `iter` и снова присваивает ему позицию недавно вставленного элемента. Этот (новый) элемент является первым. Таким образом, каждая итерация вставляет элемент перед первым элементом в списке.

Применение функций `emplace()`



Новый стандарт вводит три новых функции-члена — `emplace_front()`, `emplace()` и `emplace_back()`, которые создают элементы, а не копируют. Они соответствуют функциям `push_front()`, `insert()` и `push_back()`, позволяющим помещать элемент в начало контейнера, перед указанной позицией или в конец контейнера соответственно.

Когда происходит вызов функции-члена `insert()` или `push()`, им передается объект типа элемента для копирования в контейнер. Когда происходит вызов функции `emplace()`, ее аргументы передаются конструктору типа элемента, который создает элемент непосредственно в

области, контролируемой контейнером. Предположим, например, что контейнер с содержит элементы типа `Sales_data` (см. раздел 7.1.4):

```
// создает объект класса Sales_data в конце
контейнера с
// использует конструктор класса Sales_data с тремя
аргументами
c.emplace_back("978-05903534 03", 25, 15.99);
// ошибка: нет версии функции push_back(),
получающей три аргумента
c.push_back("978-0590353403", 25, 15.99);
// ok: создается временный объект класса Sales_data
для передачи
// функции push_back()
c.push_back(Sales_data("978-0590353403",
25,
15.99));
```

Вызов функции `emplace_back()` и второй вызов функции `push_back()` создают новые объекты класса `Sales_data`. При вызове функции `emplace_back()` этот объект создается непосредственно в области, контролируемой контейнером. Вызов функции `push_back()` создает локальный временный объект, который помещается в контейнер.

Аргументы функции `emplace()` зависят от типа элемента, они должны соответствовать конструктору типа элемента:

```
// iter указывает на элемент класса Sales_data
контейнера с
c.emplace_back(); // использует стандартный
конструктор
// класса Sales_data
c.emplace(iter, "999-99999999"); // используется
Sales_data(string)
// использует конструктор класса Sales_data,
получающий ISBN,
// количество и цену
c.emplace_front("978-0590353403", 25, 15.99);
```



Функция `emplace()` создает элементы контейнера. Ее аргументы должны

соответствовать конструктору типа элемента.

Упражнения раздела 9.3.1

Упражнение 9.18. Напишите программу чтения последовательности строк со стандартного устройства ввода в контейнер `deque`. Для записи элементов в контейнер `deque` используйте итераторы и цикл.

Упражнение 9.19. Перепишите программу из предыдущего упражнения, чтобы использовался контейнер `list`. Перечислите необходимые изменения.

Упражнение 9.20. Напишите программу, копирующую элементы списка `list<int>` в две двухсторонние очереди, причем нечетные элементы должны копироваться в один контейнер `deque`, а четные в другой.

Упражнение 9.21. Объясните, как цикл из пункта «Применение возвращаемого значения функции `insert()`», использующий возвращаемое значение функции `insert()` и добавляющий элементы в список, работал бы с вектором вместо списка.

Упражнение 9.22. С учетом того, что `iv` является вектором целых чисел, что не так со следующей программой? Как ее можно исправить?

```
vector<int>::iterator iter = iv.begin(),
mid = iv.begin() + iv.size() / 2;
while (iter != mid)
if (*iter == some_val)
iv.insert(iter, 2 * some_val);
```



9.3.2. Доступ к элементам

В табл. 9.6 приведен список функций, которые можно использовать для доступа к элементам последовательного контейнера. Если в контейнере нет элементов, функции доступа неприменимы.

У каждого последовательного контейнера, включая `array`, есть функция-член `front()`, и у всех, кроме `forward_list`, есть также функция-член `back()`. Эти функции возвращают ссылку на первый и последний элементы соответственно:

```
// перед обращением к значению итератора
удостовериться, что
```

```

// элемент существует, либо вызвать функции front( )
и back()
if (!c.empty()) {
    // val и val2 - копии значений первого элемента в
контейнере c
    auto val = *c.begin(), val2 = c.front();
    // val3 и val4 - копии значений последнего
элемента в контейнере c
    auto last = c.end();
    auto val3 = *(--last); // невозможен декремент
итератора forward_list
    auto val4 = c.back(); // не поддерживается
forward_list
}

```

Таблица 9.6. Функции доступа к элементам последовательного контейнера

Функция at() и оператор индексирования допустимы только для контейнеров string, vector, deque и array. Функция back() недопустима для контейнера forward_list.	
c. back()	Возвращает ссылку на последний элемент контейнера c, если он не пуст
c. front()	Возвращает ссылку на первый элемент контейнера c, если он не пуст
c[n]	Возвращает ссылку на элемент, индексированный целочисленным беззнаковым значением n. Если n >= c.size(), результат непредсказуем
c. at(n)	Возвращает ссылку на элемент по индексу n. Если индекс находится вне диапазона, передает исключение out_of_range

Эта программа получает ссылки на первый и последний элементы контейнера с двумя разными способами. Прямой подход — обращение к функциям front() и back(). Косвенный подход получения ссылки на тот же элемент подразумевает обращение к значению итератора, возвращенного функцией begin(), или декремент с последующим обращением к значению итератора, возвращенного функцией end().

В этой программе примечательны два момента: поскольку возвращенный функцией end() итератор указывает на элемент, следующий после последнего, для доступа к последнему элементу контейнера применяется декремент полученного итератора. Вторым очень важным моментом является необходимость удостовериться в том, что

контейнер с не пуст, перед вызовом функций `front()` и `back()` или обращением к значению итераторов, возвращенных функциями `begin()` и `end()`. Если контейнер окажется пустым, все выражения в блоке операторов `if` будут некорректны.



Вызов функций `front()` или `back()` для пустого контейнера, равно как и использование индекса, выходящего за диапазон существующих элементов, является серьезной ошибкой.

Функции-члены, обращающиеся к элементам в контейнере (т.е. функции `front()`, `back()`, `at()` и индексирование), возвращают ссылки. Если контейнер является константным объектом, возвращается ссылка на константу. Если контейнер не константа, возвращается обычная ссылка, которую можно использовать для изменения значения выбранного элемента:

```
if (!c.empty()) {  
    c.front() = 42;           // присвоить 42 первому  
    элементу в контейнере c  
    auto &v = c.back();      // получить ссылку на  
    последний элемент  
    v = 1024;                // изменить элемент в  
    контейнере c  
    auto v2 = c.back();      // v2 не ссылка; это копия  
    c.back()  
    v2 = 0;                  // это не изменит элемент в  
    контейнере c  
}
```

Как обычно, если ключевое слово `auto` применяется для определения переменной, сохраняющей значение, возвращаемые одной из этих функций, и эту переменную предстоит использовать для изменения элемента, то определять эту переменную следует как ссылочный тип.

Индексация и безопасный произвольный доступ

Контейнеры, предоставляющие быстрый произвольный доступ (`string`, `vector`, `deque` и `array`), предоставляют также оператор индексирования (см. раздел 3.3.3). Как уже упоминалось, оператор

индексирования получает индекс и возвращает ссылку на элемент в этой позиции контейнера. Индекс не должен выходить за диапазон элементов (т.е. больше или равен 0 и меньше размера контейнера). Допустимость индекса должен обеспечить разработчик; оператор индексирования не проверяет принадлежность индекса диапазону. Использование для индекса значения вне диапазона является серьезной ошибкой, но компилятор ее не обнаружит.

Если необходимо гарантировать допустимость индекса, вместо него можно использовать функцию-член `at()`. Она действует как оператор индексирования, но если индекс недопустим, то она передает исключение `out_of_range` (см. раздел 5.6):

```
vector<string> svec; // пустой вектор
cout << svec[0];      // ошибка времени выполнения:
вектор svec
                           // еще не имеет элементов!
cout << svec.at(0);    // передает исключение
out_of_range
```

Упражнения раздела 9.3.2

Упражнение 9.23. Какими были бы значения переменных `val2`, `val3` и `val4`, в первой программе данного раздела, если бы функция `c.size()` возвращала значение 1?

Упражнение 9.24. Напишите программу, которая обращается к первому элементу вектора, используя функции `at()`, `front()` и `begin()`, а также оператор индексирования. Проверьте программу на пустом векторе.



9.3.3. Удаление элементов

Подобно тому, как существует несколько способов добавления элементов в контейнер (исключая `array`), существует несколько способов их удаления. Функции удаления перечислены в табл. 9.7.

Таблица 9.7. Функции удаления последовательных контейнеров

Эти функции изменяют размер контейнера; они не поддерживаются массивами. Контейнер `forward_list` обладает специальной версией функции `erase()`; см.

раздел 9.3.4, а функции `pop_back()` у него нет. Функция `pop_front()` недопустима для контейнеров `vector` и `string`.

<code>c.pop_back()</code>	Удаляет последний элемент контейнера <code>c</code> . Результат непредсказуем, если контейнер <code>c</code> пуст. Возвращает <code>void</code>
<code>c.pop_front()</code>	Удаляет первый элемент контейнера <code>c</code> . Результат непредсказуем, если контейнер <code>c</code> пуст. Возвращает <code>void</code>
<code>c.erase(p)</code>	Удаляет элемент, обозначенный итератором <code>p</code> . Возвращает итератор на элемент после удаленного или <i>итератор после конца</i> (<i>off-the-end iterator</i>), если итератор <code>p</code> обозначает последний элемент. Результат непредсказуем, если итератор <code>p</code> указывает на следующий элемент после последнего
<code>c.erase(b, e)</code>	Удаляет диапазон элементов, обозначенных итераторами <code>b</code> и <code>e</code> . Возвращает итератор на элемент после последнего удаленного или после последнего элемента контейнера, если итератор <code>e</code> указывал на последний элемент
<code>c.clear()</code>	Удаляет все элементы контейнера <code>c</code> . Возвращает <code>void</code>



ВНИМАНИЕ

Функции-члены удаления элементов не проверяют свои аргументы. Разработчик должен сам позаботиться о проверке наличия элементов перед их удалением.

Применение функций `pop_front()` и `pop_back()`

Функции `pop_front()` и `pop_back()` удаляют, соответственно, первый и последний элементы контейнера. Векторы и строки функциями `push_front()` и `pop_front()` не обладают. У контейнера `forward_list` также нет функции `pop_back()`. Подобно функциям-членам доступа к элементам, эти функции не применимы к пустому контейнеру.

Удалив соответствующий элемент, эти функции возвращают тип `void`. Если необходимо извлечь элемент, то следует сохранить его значение перед удалением:

```
while (!ilist.empty()) {
    process(ilist.front());      // действия с текущей
вершиной списка ilist
    ilist.pop_front();          // готово; удалить первый
элемент
```

}



Удаление элементов в любой позиции, кроме начала или конца двухсторонней очереди, объявляет недействительными все итераторы, ссылки и указатели. В векторе и строке недействительными оказываются итераторы, ссылки и указатели на элементы, расположенные после удаленного элемента.

Удаление элемента в середине контейнера

Функции-члены удаления удаляют элемент (элементы) в указанной позиции контейнера. Можно удалить как отдельный элемент, обозначенный итератором, так и диапазон элементов, отмеченных парой итераторов. Обе формы функции `erase()` возвращают итератор на область *после* последнего удаленного элемента.

В качестве примера рассмотрим следующий цикл, удаляющий нечетные элементы списка:

```
list<int> lst = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;
auto it = lst.begin() ;
while (it != lst.end())
    if (*it % 2) // если элемент является
        нечетным
        it = lst.erase(it); // удалить его
    else
        ++it;
```

На каждой итерации проверяется нечетность текущего элемента. Если это так, то данный элемент удаляется, а итератор `it` устанавливается на следующий элемент после удаленного. Если элемент `*it` четный, осуществляется приращение итератора `it`, чтобы при следующей итерации он указывал на следующий элемент.

Удаление нескольких элементов

Версия функции `erase()` с парой итераторов позволяет удалить диапазон элементов:

```
// удалить диапазон элементов между двумя
итераторами
// возвращает итератор на элемент сразу после
```

последнего удаленного

```
elem1 = slist.erase(elem1, elem2); // после вызова  
elem1 == elem2
```

Итератор `elem1` указывает на первый удаляемый элемент, а итератор `elem2` — на следующий после последнего удаляемого.

Чтобы удалить все элементы в контейнере, можно либо вызвать функцию `clear()`, либо перебирать итераторы от возвращаемого функцией `begin()` до `end()` и передавать их функции `erase()`:

```
slist.clear(); // удалить все элементы в контейнере  
slist.erase(slist.begin(), slist.end()); //  
эквивалент
```

Упражнения раздела 9.3.3

Упражнение 9.25. Что будет, если в программе, где удалялся диапазон элементов, итераторы `elem1` и `elem2` равны? Что если итератор `elem2` или оба итератора (`elem1` и `elem2`) являются итератором после конца?

Упражнение 9.26. Используя приведенное ниже определение массива `ia`, скопируйте его содержимое в вектор и в список. Используя версию функции `erase()` для одного итератора, удалите из списка элементы с нечетными значениями, а из вектора — с четными.

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 };
```



9.3.4. Специализированные функции контейнера forward_list

Чтобы лучше понять, почему у контейнера `forward_list` есть специальные версии функций добавления и удаления элементов, рассмотрим, что происходит при удалении элемента из односвязного списка. Как показано на рис. 9.1, удаление элемента изменяет связи в последовательности. В данном случае удаление элемента `elem3` изменяет связь элемента `elem2`; элемент `elem2` указывал на элемент `elem3`, но после удаления элемента `elem3` элемент `elem2` указывает на элемент `elem4`.



Удаление элемента elem3 изменяет элемент elem2



Рис. 9.1. Специализированные функции контейнера `forward_list`

При добавлении или удалении элемента у элемента перед ним будет другой последователь. Чтобы добавить или удалить элемент, необходимо обратиться к его предшественнику и изменить его ссылку. Однако контейнер `forward_list` — это односвязный список. В односвязном списке нет простого способа доступа к предыдущему элементу. Поэтому функции добавления и удаления элементов в контейнере `forward_list` работают, изменения элемент после указанного. Таким образом, у нас всегда есть доступ к элементам, на которые влияет изменение.

Поскольку эти функции ведут себя отлично от функций других контейнеров, класс `forward_list` не определяет функции-члены `insert()`, `emplace()` и `erase()`. Вместо них используются функции-члены `insert_after()`, `emplace_after()` и `erase_after()` (перечислены в табл. 9.8). На рисунке выше, например, для удаления элемента `elem3` использовалась бы функция `erase_after()` для итератора, обозначающего элемент `elem2`. Для реализации этих операций класс `forward_list` определяет также функцию `before_begin()`, которая возвращает *итератор после начала* (*off-the-beginning iterator*). Этот итератор позволяет добавлять или удалять элементы *после* несуществующего элемента перед первым в списке.

Таблица 9.8. Функции вставки и удаления элементов контейнера `forward_list`

<code>lst.before_begin()</code> <code>lst.cbefore_begin()</code>	Возвращает итератор, обозначающий несуществующий элемент непосредственно перед началом списка. К значению этого итератора обратиться нельзя. Функция <code>cbefore_begin()</code> возвращает итератор <code>const_iterator</code>
<code>lst.insert_after(p, t)</code> <code>lst.insert_after(p, n, t)</code> <code>lst.insert_after(p, b, e)</code> <code>lst.insert_after(p, il)</code>	Вставляет элемент (элементы) после обозначенного итератором <code>p</code> . <code>t</code> — это объект, <code>n</code> — количество, <code>b</code> и <code>e</code> — обозначающие диапазон итераторы (они не должны принадлежать контейнеру <code>lst</code>), <code>il</code> — список, заключенный в скобки. Возвращает итератор на <i>последний</i> вставленный элемент. Если диапазон пуст,

	возвращается итератор <i>p</i> . Если <i>p</i> — итератор после конца, результат будет непредсказуемым
<i>emplace_after(p, args)</i>	Параметр <i>args</i> используется для создания элементов после обозначенного итератором <i>p</i> . Возвращает итератор на новый элемент. Если <i>p</i> — итератор после конца, результат будет непредсказуемым
<i>lst.erase_after(p)</i> <i>lst.erase_after(b, e)</i>	Удаляет элемент после обозначенного итератором <i>p</i> или диапазоном элементов после обозначенного итератором <i>b</i> и до, но не включая обозначенного итератором <i>e</i> . Возвращает итератор на элемент после удаленного или на элемент после конца контейнера. Если <i>p</i> — итератор после конца или последний элемент контейнера, результат будет непредсказуемым

При добавлении или удалении элементов в контейнер *forward_list* следует обратить внимание на два итератора: на проверяемый элемент и на элемент, предшествующий ему. В качестве примера перепишем приведенный выше цикл, удалявший из списка нечетные элементы, так, чтобы использовался контейнер *forward_list*:

```
forward_list<int> flst = {0,1,2,3,4,5,6,7,8,9};
auto prev = flst.before_begin(); // обозначает
элемент "перед началом"
                                                // контейнера flst
auto curr = flst.begin(); // обозначает первый
элемент контейнера flst
while (curr != flst.end()) { // пока есть элементы
для обработки
    if (*curr % 2)                                // если элемент
нечетный
        curr = flst.erase_after(prev); // удалить его и
переместить curr
    else {
        prev = curr; // переместить итератор на следующий
элемент
        ++curr;      // и один перед следующим элементом
    }
}
}

Here the iterator curr marks the element being checked, and the iterator prev — the element before curr. The iterator curr initializes the call to the begin() function, so the first iteration checks the first element for evenness. The iterator prev initializes the call to the erase_after() function.
```

Здесь итератор *curr* обозначает проверяемый элемент, а итератор *prev* — элемент перед *curr*. Итератор *curr* инициализирует вызов функции *begin()*, чтобы первая итерация проверила на четность первый элемент. Итератор *prev* инициализирует вызов функции

`before_begin()`, который возвращает итератор на несуществующий элемент непосредственно перед `curr`.

Когда находится нечетный элемент, итератор `prev` передается функции `erase_after()`. Этот вызов удаляет элемент после обозначенного итератором `prev`; т.е. элемент, обозначенный итератором `curr`. Итератору `curr` присваивается значение, возвращенное функцией `erase_after()`. В результате он обозначит следующий элемент последовательности, а итератор `prev` останется неизменным; он все еще обозначает элемент перед (новым) значением итератора `curr`. Если обозначенный итератором `curr` элемент не является нечетным, то в части `else` оба итератора перемещаются на следующий элемент.

Упражнения раздела 9.3.4

Упражнение 9.27. Напишите программу для поиска и удаления нечетных элементов в контейнере `forward_list<int>`.

Упражнение 9.28. Напишите функцию, получающую контейнер `forward_list<string>` и два дополнительных аргумента типа `string`. Функция должна находить первую строку и вставлять вторую непосредственно после первой. Если первая строка не найдена, то вставьте вторую строку в конец списка.

9.3.5. Изменение размеров контейнера

Для изменения размера контейнера, за исключением массива, можно использовать функцию `resize()`, представленную в табл. 9.9. Если текущий размер больше затребованного, элементы удаляются с конца контейнера; если текущий размер меньше нового, элементы добавляются в конец контейнера:

```
list<int> ilist( 10, 42 ); // 10 целых чисел со  
значением 42  
ilist.resize( 15 ); // добавляет 5 элементов  
со значением 0 // в конец списка ilist  
ilist.resize( 25, -1 ); // добавляет 10 элементов  
со значением -1 // в конец списка ilist  
ilist.resize( 5 ); // удаляет 20 элементов от  
конца списка ilist
```

Функция `resize()` получает необязательный аргумент — значение элемента, используемое для инициализации всех добавляемых элементов. Если этот аргумент отсутствует, добавленные элементы инициализируются значением по умолчанию (см. раздел 3.3.1). Если контейнер хранит элементы типа класса и функция `resize()` добавляет элементы, то либо следует предоставить инициализатор, либо тип элемента должен иметь стандартный конструктор.

Таблица 9.9. Функции размера последовательного контейнера

За исключением массива	
<code>c.resize(n)</code>	Измените размеры контейнера <code>c</code> так, чтобы у него было <code>n</code> элементов. Если <code>n < c.size()</code> , то лишние элементы отбрасываются. Если следует добавить новые элементы, они инициализируются значением по умолчанию
<code>c.resize(n, t)</code>	Измените размеры контейнера <code>c</code> так, чтобы у него было <code>n</code> элементов. Все добавляемые элементы получат значение <code>t</code>



Если функция `resize()` сокращает контейнер, то итераторы, ссылки и указатели на удаленные элементы окажутся некорректными; выполнение функции `resize()` для контейнеров `vector`, `string` и `deque` может сделать некорректными все итераторы, указатели и ссылки.

Упражнения раздела 9.3.5

Упражнение 9.29. Если контейнер `vec` содержит 25 элементов, то что делает выражение `vec.resize(100)`? Что если затем последует вызов `vec.resize(10)`?

Упражнение 9.30. Какие ограничения (если они есть) налагает использование функции `resize()` с одиночным аргументом, имеющим тип элемента?



9.3.6. Некоторые операции с контейнерами делают итераторы недопустимыми

Функции, добавляющие и удаляющие элементы из контейнера, могут сделать некорректными указатели, ссылки или итераторы на его элементы. Некорректными считаются те указатели, ссылки и итераторы, которые больше не указывают на элемент. Использование некорректного указателя, ссылки или итератора является серьезной ошибкой, последствия которой, вероятно, будут схожи с использованием неинициализированного указателя (см. раздел 2.3.2, стр. 89).

После операции добавления элементов в контейнер возможно следующее.

- Итераторы, указатели и ссылки на элементы вектора или строки становятся недопустимыми после повторного резервирования пространства контейнера. Если повторного резервирования не было, ссылки на элементы перед позицией вставки остаются допустимыми, а на элементы после позиции вставки — нет.

- Итераторы, указатели и ссылки на элементы двухсторонней очереди становятся недопустимыми после добавления элементов в любую позицию кроме начала или конца. При добавлении в начало или в конец недопустимыми становятся только итераторы, а ссылки и указатели на существующие элементы — нет.

Нет ничего удивительного в том, что после удаления элементов из контейнера итераторы, указатели и ссылки на удаленные элементы становятся недопустимыми. В конце концов, этих элементов больше нет.

- У контейнеров `list` и `forward_list` все остальные итераторы, ссылки и указатели (включая итераторы после конца и перед началом) остаются допустимыми.

- У контейнера `deque` все остальные итераторы, ссылки и указатели становятся недопустимыми, если удалены элементы в любой позиции, кроме начала или конца. Если удаляются элементы в конце, итератор после конца становится недопустимым, но другие итераторы, ссылки и указатели остаются вполне допустимыми. То же относится к удалению из начала.

- У контейнеров `vector` и `string` все остальные итераторы, ссылки и указатели на элементы перед позицией удаления остаются допустимыми. При удалении элементов итератор после конца всегда оказывается недопустимым.



ВНИМАНИЕ

Использование недопустимого итератора, указателя или ссылки является

серьезной ошибкой, которая проявится во время выполнения программы.

Совет. Контроль итераторов

При использовании итератора (или ссылки, или указателя на элемент контейнера) имеет смысл минимизировать ту часть программы, где итератор обязан оставаться допустимым.

Поскольку код, добавляющий или удаляющий элементы из контейнера, может сделать итераторы недопустимыми, необходимо позаботиться о переустановке итераторов соответствующим образом после каждой операции, которая изменяет контейнер. Это особенно важно для контейнеров `vector`, `string` и `deque`.



Создание циклов, которые изменяют контейнер

Циклы, добавляющие или удаляющие элементы из контейнеров `vector`, `string` или `deque`, должны учитывать тот факт, что итераторы, ссылки и указатели могут стать недопустимыми. Программа должна гарантировать, что итератор, ссылка или указатель обновляется на каждом цикле. Если цикл использует функцию `insert()` или `erase()`, обновить итератор довольно просто. Они возвращают итераторы, которые можно использовать для переустановки итератора:

```
// бесполезный цикл, удаляющий четные элементы и
вставляющий дубликаты
// нечетных
vector<int> vi = {0,1,2,3,4,5,6,7,8,9};
auto iter = vi.begin(); // поскольку vi изменяется,
используется
// функция begin(), а не
cbegin()
while (iter != vi.end()) {
    if (*iter % 2) {
        iter = vi.insert(iter, *iter); // дублирует
текущий элемент
        iter +=2; // переместить
через элемент
    } else
        iter = vi.erase(iter); // удалить четные элементы
```

```
// не перемещать итератор; iter обозначает элемент
после
// удаленного
}
```

Эта программа удаляет четные элементы и дублирует соответствующие нечетные. После вызова функций `insert()` и `erase()` итератор обновляется, поскольку любая из них способна сделать итератор недопустимой.

После вызова функции `erase()` никакой необходимости в приращении итератора нет, поскольку возвращенный ею итератор обозначает следующий элемент в последовательности. После вызова функции `insert()` итератор увеличивается на два. Помните, функция `insert()` осуществляет вставку перед указанной позицией и возвращает итератор на вставленный элемент. Таким образом, после вызова функции `insert()` итератор `iter` обозначает элемент (недавно добавленный) перед обрабатываемым. Приращение на два осуществляется для того, чтобы перескочить через добавленный и только что обработанный элементы. Это перемещает итератор на следующий необработанный элемент.



Не храните итератор, возвращенный функцией `end()`

При добавлении или удалении элементов в контейнер `vector` или `string` либо при добавлении или удалении элементов в любую, кроме первой, позицию контейнера `deque` возвращенный функцией `end()` итератор всегда будет недопустимым. Потому циклы, которые добавляют или удаляют элементы, всегда должны вызывать функцию `end()`, а не использовать хранимую копию. Частично поэтому стандартные библиотеки C++ реализуют обычно функцию `end()` так, чтобы она выполнялась очень быстро.

Рассмотрим, например, цикл, который обрабатывает каждый элемент и добавляет новый элемент после исходного. Цикл должен игнорировать добавленные элементы и обрабатывать только исходные. После каждой вставки итератор позиционируется так, чтобы обозначить следующий исходный элемент. Если попытаться "оптимизировать" цикл, сохраняя итератор, возвращенный функцией `end()`, то будет беда:

```
// ошибка: поведение этого цикла непредсказуемо
```

```

auto begin = v.begin(),
      end = v.end(); // плохая идея хранить значение
итератора end
while (begin != end) {
    // некоторые действия
    // вставить новое значение и переприсвоить
итератор begin, который
    // в противном случае окажется недопустимым
    ++begin; // переместить begin, поскольку вставка
необходима после
    // этого элемента
begin = v.insert(begin, 42); // вставить новое
значение
++begin; // переместить begin за только что
добавленный элемент
}

```

Поведение этого кода непредсказуемо. На многих реализациях получится бесконечный цикл. Проблема в том, что возвращенное функцией `end()` значение хранится в локальной переменной `end`. В теле цикла добавляется элемент. Добавление элемента делает итератор, хранимый в переменной `end`, недопустимым. Этот итератор не указывает ни на какой элемент в контейнере `v`, ни на следующий после его конца.



Не кешируйте возвращаемый функцией `end()` итератор в циклах, которые вставляют или удаляют элементы в контейнере `deque`, `string` или `vector`.

Вместо того чтобы хранить итератор `end`, его следует повторно вычислять после каждой вставки:

```

// существенно безопасный: повторно вычислять end
после каждого
// добавления/удаления элементов
while (begin != v.end()) {
    // некоторые действия
    ++begin; // переместить begin, поскольку вставка
необходима после

```

```

    // этого элемента
begin = v.insert( begin, 42); // вставить новое
значение
++begin; // переместить begin за только что
добавленный элемент
}

```

Упражнения раздела 9.3.6

Упражнение 9.31. Программа из пункта «Создание циклов, которые изменяют контейнер», удаляющая четные и дублирующая нечетные элементы, не будет работать с контейнером `list` или `forward_list`. Почему? Переделайте программу так, чтобы она работала и с этими типами тоже.

Упражнение 9.32. Будет ли допустим в указанной выше программе следующий вызов функции `insert()`? Если нет, то почему?

```
iter = vi.insert( iter, *iter++);
```

Упражнение 9.33. Что будет, если в последнем примере этого раздела не присваивать переменной `begin` результат вызова функции `insert()`? Напишите программу без этого присвоения, чтобы убедиться в правильности своего предположения.

Упражнение 9.34. Учитывая, что `vi` является контейнером целых чисел, содержащим четные и нечетные значения, предскажите поведение следующего цикла. Проанализировав этот цикл, напишите программу, чтобы проверить правильность своих ожиданий.

```

iter = vi.begin();
while (iter != vi.end())
if (*iter % 2)
    iter = vi.insert( iter, *iter);
++iter;

```



9.4. Как увеличивается размер вектора

Для обеспечения быстрого произвольного доступа элементы вектора хранятся последовательно — каждый элемент рядом с предыдущим. Как правило, нас не должно заботить то, как реализован библиотечный тип; достаточно знать, как его использовать. Но в случае векторов и строк реализация частично просачивается в интерфейс.

С учетом того, что элементы последовательны и размер контейнера гибок, рассмотрим происходящее при добавлении элемента в вектор или строку: если для нового элемента нет места, контейнер не сможет просто добавить элемент в некую другую область памяти — элементы должны располагаться последовательно. Поэтому контейнер должен зарезервировать новую область памяти, достаточную для содержания уже существующих элементов, плюс новый элемент, а затем переместить элементы из старой области в новую, добавить новый элемент и освободить старую область памяти. Если бы вектор осуществлял резервирование и освобождение памяти при каждом добавлении элемента, его работа была бы неприемлемо медленной.

Во избежание дополнительных затрат конструкторы библиотечных контейнеров используют стратегию, сокращающую количество повторных резервирований. При необходимости выделения новой области памяти реализации классов `vector` и `string` обычно резервируют больший объем, чем необходимо в данный момент. Контейнер хранит его в резерве и использует для размещения новых элементов при их добавлении. Поэтому нет никакой необходимости в повторном резервировании места для контейнера при каждом добавлении нового элемента.

Эта стратегия резервирования существенно эффективней таковой при каждой необходимости добавления нового элемента. Фактически ее производительность настолько высока, что на практике вектор обычно растет эффективней, чем список или двухсторонняя очередь, хотя вектор должен еще перемещать свои элементы при каждом повторном резервировании памяти.

Функции-члены управления емкостью

Типы `vector` и `string` предоставляют описанные в табл. 9.10

функции-члены, позволяющие взаимодействовать с частью реализации, относящейся к резервированию памяти. Функция `capacity()` сообщает количество элементов, которое контейнер может создать прежде, чем ему понадобится занять больший объем памяти. Функция `reserve()` позволяет задать количество резервируемых элементов.

Таблица 9.10. Управление размером контейнера

Функция <code>shrink_to_fit()</code> допустима только для контейнеров <code>vector</code> , <code>string</code> и <code>deque</code> . Функции <code>capacity()</code> и <code>reserve()</code> допустимы только для контейнеров <code>vector</code> и <code>string</code> .	
<code>c.shrink_to_fit()</code>	Запрос на уменьшение емкости в соответствии с размером
<code>c.capacity()</code>	Количество элементов, которое может иметь контейнер с прежде, чем понадобится повторное резервирование
<code>c.reserve(n)</code>	Резервирование места по крайней мере для <code>n</code> элементов



Функция `reserve()` не изменяет количество элементов в контейнере; она влияет только на объем памяти, предварительно резервируемой вектором.

Вызов функции `reserve()` изменяет емкость вектора, только если требуемое пространство превышает текущую емкость. Если требуемый размер больше текущей емкости, функция `reserve()` резервирует по крайней мере столько места, сколько затребовано (или несколько больше).

Если требуемый размер меньше или равен существующей емкости, функция `reserve()` ничего не делает. В частности, вызов функции `reserve()`, при размере меньшем, чем емкость, не приведет к резервированию контейнером памяти. Таким образом, после вызова функции `reserve()` емкость будет больше или равна переданному ей аргументу.

В результате вызов функции `reserve()` никогда не сократит объем контейнера. Точно так же функция-член `resize()` (см. раздел 9.3.5) изменяет только количество элементов контейнера, а не его емкость. Функцию `resize()` нельзя использовать для сокращения объема память, которую контейнер держит в резерве.

В новой библиотеке есть функция `shrink_to_fit()`, позволяющая запросить контейнеры `deque`, `vector` или `string` освободить неиспользуемую память. Вызов этой функции означает, что никакой резервной емкости больше не нужно. Однако реализация имеет право проигнорировать этот запрос. Нет никакой гарантии, что вызов функции `shrink_to_fit()` освободит память.

Функции-члены `capacity()` и `size()`

Очень важно понимать различие между *емкостью* (`capacity`) и *размером* (`size`). Размер — это количество элементов, хранящихся в контейнере в настоящий момент, а емкость — это количество элементов, которое контейнер может содержать, не прибегая к следующей операции резервирования памяти. Следующий код иллюстрирует взаимосвязь размера и емкости:

```
vector<int> ivec;
// размер нулевой; емкость зависит от реализации
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
// присвоить вектору ivec 24 элемента
for (vector<int>::size_type ix = 0; ix != 24; ++ix)
    ivec.push_back(ix);

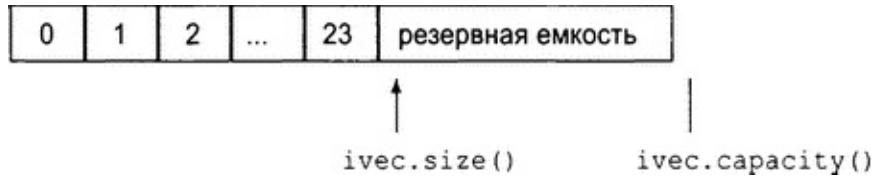
// размер 24; емкость равна или больше 24, согласно
реализации
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
```

При запуске на компьютере автора эта программа отобразила следующий результат:

```
ivec: size: 0 capacity: 0
ivec: size: 24 capacity: 32
```

Как известно, пустой вектор имеет нулевой размер, вполне очевидно, что библиотека для пустого вектора также устанавливает нулевую емкость. При добавлении элементов в вектор его размер составляет количество добавленных элементов. Емкость будет, по крайней мере совпадать с размером, но может быть и больше. Конкретный объем резервной емкости зависит от реализации библиотеки. В данной конкретной реализации добавление 24 элементов по одному приводит к созданию емкости 32.

Визуально текущее состояние вектора `ivec` можно представить так:



Теперь при помощи функции `reserve()` можно зарезервировать дополнительное пространство.

```
iVec.reserve(50); // задать емкость 50 элементов  
(можно и больше)
```

// размер будет 24, а емкость - 50 или больше, если так определено

// в реализации

```
cout << "iVec: size: " << iVec.size()  
     << " capacity: " << iVec.capacity() << endl;
```

Вывод свидетельствует о том, что вызов функции `reserve()` зарезервировал именно столько места, сколько было запрошено:

```
iVec: size: 24 capacity: 50
```

Эту резервную емкость можно впоследствии израсходовать следующим образом:

```
// добавить элементы, чтобы исчерпать резервную  
емкость
```

```
while (iVec.size() != iVec.capacity())
```

```
    iVec.push_back(0);
```

// емкость не изменилась, размер и емкость теперь равны

```
cout << "iVec: size: " << iVec.size()
```

```
     << " capacity: " << iVec.capacity() << endl;
```

Результат свидетельствует, что в настоящий момент резервная емкость исчерпана, а размер и емкость равны.

```
iVec: size: 50 capacity: 50
```

Поскольку использовалась только резервная емкость, в повторном резервировании нет никакой потребности. Фактически, пока не превышена существующая емкость вектора, никакой необходимости в перераспределении его элементов нет.

Если теперь добавить в вектор новый элемент, то последует повторное резервирование памяти.

```
iVec.push_back(42); // добавить еще один элемент
```

// размер будет 51, а емкость 51 или больше, если так определено

```
// в реализации
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
```

Результат выполнения этой части программы имеет следующий вид:
ivec: size: 51 capacity: 100

Он свидетельствует о том, что в данной реализации класса `vector` использована стратегия удвоения текущей емкости при каждом резервировании новой области памяти.

По мере необходимости можно вызвать функцию `shrink_to_fit()`, запрашивающую освобождение и возвращение операционной системе памяти, ненужной для текущего размера контейнера:

```
ivec.shrink_to_fit(); // запрос на возвращение памяти
```

```
// размер остался неизменным; емкость определена реализацией
```

```
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
```

Вызов функции `shrink_to_fit()` является только запросом; нет никакой гарантии того, что память будет действительно возвращена.



Каждая реализация контейнера `vector` может использовать собственную стратегию резервирования. Однако резервирование новой памяти не должно происходить, пока его емкость не исчерпана.

Вектор может начать повторное резервирование только после выполнения пользователем операции вставки, когда размер равен емкости, вызова функции `resize()` или `reserve()` со значением, превышающим текущую емкость. Количество памяти, резервируемое свыше указанного объема, зависит от реализации.

Каждая реализация обязана следовать стратегии, гарантирующей эффективное использование функции `push_back()` при добавлении элементов в вектор. С технической точки зрения время создания n элементов вектора составляет время выполнения функции `push_back()` для первоначально пустого вектора, умноженное на n .

Упражнения раздела 9.4

Упражнение 9.35. Объясните различие между емкостью вектора и его размером.

Упражнение 9.36. Может ли контейнер иметь емкость, меньшую, чем его размер?

Упражнение 9.37. Почему контейнеры `list` и `array` не имеют функции-члена `capacity()`?

Упражнение 9.38. Напишите программу, позволяющую исследовать рост вектора в библиотеке, которую вы используете.

Упражнение 9.39. Объясните, что выполняет следующий фрагмент программы:

```
vector<string> svec;
svec.reserve(1024);
string word;
while (cin >> word)
    svec.push_back(word);
svec.resize(svec.size() + svec.size() / 2);
```

Упражнение 9.40. Если программа в предыдущем упражнении читает 256 слов, какова ее вероятная емкость после вызова функции `resize()`? Что, если она читает 512, 1 000 или 1 048 слов?

9.5. Дополнительные операции со строками

Кроме операций, общих для всех последовательных контейнеров, тип `string` предоставляет множество дополнительных. По большей части, эти дополнительные операции либо обеспечивают взаимодействие класса `string` и символьных массивов в стиле С, либо являются дополнительными версиями функций, позволяющими использовать индексы вместо итераторов.

Библиотека `string` определяет множество функций, которые, к счастью, следуют единому шаблону. С учетом количества обсуждаемых функций этот раздел может показаться слишком громоздким при первом чтении; поэтому сначала его можно просто просмотреть. Впоследствии, имея представление о существующих видах операций, можно при необходимости вернуться и изучить подробности использования конкретной функции.



9.5.1. Дополнительные способы создания строк

В дополнение к конструкторам, описанным в разделе 3.2.1, и конструкторам, общим для всех последовательных контейнеров (см. табл. 9.3), тип `string` предоставляет еще три конструктора, описанные в табл. 9.11.

Таблица 9.11. Дополнительные способы создания строк

Все значения <code>n</code> , <code>len2</code> и <code>pos2</code> являются беззнаковыми.	
<code>string s(cp, n);</code>	Строка <code>s</code> — копия первых <code>n</code> символов из массива, на который указывает <code>cp</code> . У того массива должно быть по крайней мере <code>n</code> символов
<code>string s(s2, pos2);</code>	Строка <code>s</code> — копия символов из строки <code>s2</code> , начиная с позиции по индексу <code>pos2</code> . Если <code>pos2 > s2.size()</code> , результат непредсказуем
<code>string s(s2, pos2, len2);</code>	Строка <code>s</code> — копия <code>len2</code> символов из строки <code>s2</code> , начиная с позиции по индексу <code>pos2</code> . Если <code>pos2 > s2.size()</code> , то результат непредсказуем. Независимо от значения <code>len2</code> , копируется по крайней мере <code>s2.size() - pos2</code> символов

Конструкторы, получающие тип `string` или `const char*`,

получают дополнительные (необязательные) аргументы, позволяющие задать количество копируемых символов. При передаче строки можно также указать индекс начала копирования:

```
const char *cp = "Hello World!!!"; // массив с
нулевым символом в конце
char noNull[] = {'H', 'i'}; // без нулевого
символа в конце
string s1(cp); // копирует cp до нулевого символа;
// s1 == "Hello World!!!"
string s2(noNull, 2); // копирует два символа из
noNull; s2 == "Hi"
string s3(noNull); // непредсказуемо: noNull не
завершается null
string s4(cp + 6, 5); // копирует 5 символов,
начиная с cp[6];
// s4 == "World"
string s5(s1, 6, 5); // копирует 5 символов,
начиная с s1[6];
// s5 == "World"
string s6(s1, 6); // копирует от s1[6] до конца
s1;
// s6 == "World!!!"
string s7(s1, 6, 20); // ok, копирует только до
конца s1;
// s7 == "World!!!"
string s8(s1, 16); // передает исключение
out_of_range
```

Обычно строка создается из типа `const char*`. Массив, на который указывает указатель, должен завершаться нулевым символом; символы копируются до нулевого символа. Если передается также количество, массив не обязан заканчиваться нулевым символом. Если количество не указано и нет нулевого символа или если указанное количество больше размера массива, результат непредсказуем.

При копировании из строки можно предоставить необязательный параметр исходной позиции и количество копируемых символов. Исходная позиция должна быть меньше или равна размеру переданной строки. Если исходная позиция больше размера, то конструктор передаст исключение `out_of_range` (см. раздел 5.6). При передаче количества копирование начинается с указанной позиции. Независимо от количества запрошенных

символов, копирование осуществляется до размера строки, но не более.

Функция substr()

Функция `substr()` (представленная в табл. 9.12) возвращает копию части или всей исходной строки. Ей можно передать (необязательно) начальную позицию и количество копируемых символов:

```
string s("hello world");
string s2 = s.substr(0, 5);    // s2 = hello
string s3 = s.substr(6);      // s3 = world
string s4 = s.substr(6, 11);   // s3 = world
string s5 = s.substr(12);     // передает исключение
out_of_range
```

Если начальная позиция превышает размер исходной строки, функция `substr()` передает исключение `out_of_range` (см. раздел 5.6). Если начальная позиция в сумме с количеством копируемых символов превосходит размер строки, то копирование осуществляется только до конца строки.

Таблица 9.12. Функция substr()

<code>s.substr(pos, n)</code>	Возвращает строку, содержащую <code>n</code> символов из строки <code>s</code> , начиная с позиции <code>pos</code> . По умолчанию параметр <code>pos</code> имеет значение 0. Параметр <code>n</code> по умолчанию имеет значение, подразумевающее копирование всех символов строки <code>s</code> , начиная с позиции <code>pos</code>
--------------------------------	--

Упражнения раздела 9.5.1

Упражнение 9.41. Напишите программу, инициализирующую строку из вектора `vector<char>`.

Упражнение 9.42. Учитывая, что символы в строку следует читать по одному и заранее известно, что прочитать предстоит по крайней мере 100 символов, как можно было бы улучшить производительность программы?



9.5.2. Другие способы изменения строки

Тип `string` поддерживает операторы присвоения последовательного контейнера, а также функции `assign()`, `insert()` и `erase()` (см. раздел 9.2.5, раздел 9.3.1 и раздел 9.3.3). В нем также определены дополнительные версии функций `insert()` и `erase()`.

В дополнение к обычным версиям функций `insert()` и `erase()`, которые получают итераторы, тип `string` предоставляет их версии, получающие индекс. Индекс указывает начальный элемент для функции `erase()` или начальную позицию для функции `insert()`:

```
s.insert(s.size(), 5, '!' ); // вставить пять восклицательных знаков  
                                // в конец строки s  
s.erase(s.size() - 5, 5); // удалить последние пять символов из  
                                // строки s
```

Библиотека `string` предоставляет также версии функций `insert()` и `assign()`, получающие массивы символов в стиле С. Например, символьный массив с нулевым символом в конце можно использовать как значение, передаваемое функциям `insert()` и `assign()`:

```
const char *cp = "Stately, plump Buck";  
s.assign(cp, 7); // s == "Stately"  
s.insert(s.size(), cp + 7); // s == "Stately, plump Buck"
```

Сначала содержимое строки `s` заменяется при вызове функции `assign()`. Присваиваемые строке `s` символы — это семь символов из начала массива, на который указывает указатель `cp`. Количество запрашиваемых символов должно быть меньше или равно количеству символов массива (исключая завершающий нулевой символ), на который указывает `cp`.

Когда происходит вызов функции `insert()` для строки `s`, подразумевается вставка символов перед несуществующим элементом в позиции `s.size()`. В данном случае копируются символы, начиная с седьмого символа `cp` и до завершающего нулевого символа.

Символы для функций `insert()` и `assign()` можно также указать как исходящие из другой строки или ее подстроки:

```
string s = "some string", s2 = "some other string";  
s.insert(0, s2); // вставить копию s2 перед позицией 0 в s  
// вставить s2.size() символов из s2,  
// начиная с позиции s2[0] перед s[0]  
s.insert(0, s2, 0, s2.size());
```

Функции `append()` и `replace()`

Класс `string` определяет две дополнительные функции-члена, `append()` и `replace()`, способные изменить содержимое строки. Все эти функции описаны в табл. 9.13. Функция `append()` — упрощенный способ вставки в конец:

```
string s("C++ Primer"), s2 = s; // инициализация
строк s и s2
                                         // текстом "C++
Primer"
s.insert(s.size(), " 4th Ed."); // s == "C++ Primer
4th Ed."
s2.append(" 4th Ed."); // эквивалент: добавление "
4th Ed." к s2;
                                         // s == s2
```

Функция `replace()` — упрощенный способ вызова функций `erase()` и `insert()`:

```
// эквивалентный способ замены "4th" на "5th"
s.erase(11, 3); // s == "C++ Primer Ed."
s.insert(11, "5th"); // s == "C++ Primer 5th Ed."
// начиная с позиции 11, удалить три символа, а
затем вставить "5th"
s2.replace(11, 3, "5th"); // эквивалент: s == s2
```

В вызове функции `replace()` вставляемый текст может быть того же размера, что и удаляемый. Но можно вставить большую или меньшую строку:

```
s.replace(11, 3, "Fifth"); // s == "C++ Primer
Fifth Ed."
```

В этом вызове удаляются три символа, но вместо них вставляются пять.

Таблица 9.13. Функции изменения содержимого строки

<code>s.insert(pos, args)</code>	Вставка символов, определенных аргументом <code>args</code> , перед позицией <code>pos</code> . Позиция <code>pos</code> может быть задана индексом или итератором. Версии, получающие индекс, возвращают ссылку на строку <code>s</code> , а получающие итератор возвращают итератор, обозначающий первый вставленный символ
<code>s.erase(pos, len)</code>	Удаляет <code>len</code> символов, начиная с позиции <code>pos</code> . Если аргумент <code>len</code> пропущен, удаляет символы от позиции <code>pos</code> до конца строки <code>s</code> . Возвращает ссылку на строку <code>s</code>
<code>s.assign(args)</code>	Заменяет символы строки <code>s</code> согласно аргументу <code>args</code> . Возвращает ссылку на строку <code>s</code>

<code>s.append(args)</code>	Добавляет аргумент <code>args</code> к строке <code>s</code> . Возвращает ссылку на строку <code>s</code>
<code>s.replace(range, args)</code>	Удаляет диапазон <code>range</code> символов из строки <code>s</code> и заменяет их символами, заданными аргументом <code>args</code> . Диапазон задан либо индексом и длиной, либо парой итераторов. Возвращает ссылку на строку <code>s</code>
Аргументы <code>args</code> могут быть одним из следующих: функции <code>append()</code> и <code>assign()</code> могут использовать все формы. Стока <code>str</code> должна быть отлична от <code>s</code>, а итераторы <code>b</code> и <code>e</code> не могут принадлежать строке <code>s</code>	
<code>str</code>	Строка <code>str</code>
<code>str, pos, len</code>	До <code>len</code> символов из строки <code>str</code> , начиная с позиции <code>pos</code>
<code>cp, len</code>	До <code>len</code> символов из символьного массива, на который указывает указатель <code>cp</code>
<code>cp</code>	Завершающийся нулевым символом массив, на который указывает указатель <code>cp</code>
<code>n, c</code>	<code>n</code> копий символа <code>c</code>
<code>b, e</code>	Символы в диапазоне, указанном итераторами <code>b</code> и <code>e</code>
<i>Список инициализации</i>	Разделяемый запятыми список символов, заключенный в фигурные скобки

Аргументы <code>args</code> для функций <code>replace()</code> и <code>insert()</code> зависят от того, использован ли позиция			
<code>replace(pos, len, args)</code>	<code>replace(b, e, args)</code>	<code>insert(pos, args)</code>	<code>insert(iter, args)</code>
Да	Да	Да	Нет
Да	Нет	Да	Нет
Да	Да	Да	Нет
Да	Да	Нет	Нет
Да	Да	Да	Да
Нет	Да	Нет	Да
Нет	Да	Нет	Да

Множество способов изменить строку

Функции `append()`, `assign()`, `insert()` и `replace()`, перечисленные в табл. 9.13, имеют несколько перегруженных версий. Аргументы этих функций зависят от того, как заданы добавляемые символы и какая часть строки изменится. К счастью, у этих функций

общий интерфейс.

У функций `assign()` и `append()` нет необходимости определять изменяемые части строки: функция `assign()` всегда заменяет все содержимое строки, а функция `append()` всегда добавляет в конец строки.

Функция `replace()` предоставляет два способа определения диапазона удаления символов. Диапазон можно определить по позиции и длине или парой итераторов. Функция `insert()` предоставляет два способа определения позиции вставки: при помощи индекса или итератора. В любом случае новый элемент (элементы) вставляется перед указанным индексом или итератором.

Существует несколько способов определения символов, добавляемых в строку. Новые символы могут быть взяты из другой строки, из указателя на символ, из заключенного в фигурные скобки списка символов или как символ и как число. Когда символы исходят из строки или указателя на символ, можно передать дополнительные аргументы, указывающие, копируются ли все символы аргумента или только часть.

Не каждая функция поддерживает все версии этих аргументов. Например, нет версии функции `insert()`, получающей индекс и список инициализации. Аналогично, если необходимо определить точку вставки, используя итератор, невозможно будет впоследствии передать символьный указатель как источник для новых символов.

Упражнения раздела 9.5.2

Упражнение 9.43. Напишите функцию, получающую три строки: `s`, `oldVal` и `newVal`. Используя итераторы, а также функции `insert()`, и `erase()` замените все присутствующие в строке `s` экземпляры строки `oldVal` строкой `newVal`. Проверьте функцию на примере замены таких общепринятых сокращений, как "tho" на "though" и "thru" на "through".

Упражнение 9.44. Перепишите предыдущую функцию так, чтобы использовались индекс и функция `replace()`.

Упражнение 9.45. Напишите функцию, получающую строку, представляющую имя и две другие строки, представляющие префикс, такой, как "Mr." или "Ms.", а также суффикс, такой, как "Jr." или "III". Используя итераторы, а также функции `insert()` и `append()`, создайте новую строку с суффиксом и префиксом, добавленным к имени.

Упражнение 9.46. Перепишите предыдущее упражнение, используя на

сей раз позицию, длину и функцию `insert()`.



9.5.3. Операции поиска строк

Класс `string` предоставляет шесть вариантов функций поиска, у каждой из которых есть четыре перегруженных версии. Функции-члены поиска и их аргументы описаны в табл. 9.14. Каждая из них возвращает значение типа `string::size_type`, являющееся индексом найденного элемента. Если соответствие не найдено, функции возвращают статический член (см. раздел 7.6) по имени `string::npos`. Библиотека определяет значение `npos` как `-1` типа `const string::size_type`. Поскольку `npos` имеет беззнаковый тип, это означает, что значение `npos` соответствует наибольшему возможному размеру, который может иметь любая строка (см. раздел 2.1.2).

Таблица 9.14. Строковые функции поиска

Функции поиска возвращают индекс искомого символа или значение <code>npos</code> , если искомое не найдено	
<code>s.find(args)</code>	Ищет первое местоположение аргумента <code>args</code> в строке <code>s</code>
<code>s.rfind(args)</code>	Ищет последнее местоположение аргумента <code>args</code> в строке <code>s</code>
<code>s.find_first_of(args)</code>	Ищет первое местоположение любого символа аргумента <code>args</code> в строке <code>s</code>
<code>s.find_last_of(args)</code>	Ищет последнее местоположение любого символа аргумента <code>args</code> в строке <code>s</code>
<code>s.find_first_not_of(args)</code>	Ищет первое местоположение символа в строке <code>s</code> , который отсутствует в аргументе <code>args</code>
<code>s.find_last_not_of(args)</code>	Ищет последнее местоположение символа в строке <code>s</code> , который отсутствует в аргументе <code>args</code>
Аргумент <code>args</code> может быть следующим	
<code>c, pos</code>	Поиск символа <code>c</code> , начиная с позиции <code>pos</code> в строке <code>s</code> . По умолчанию <code>pos</code> имеет значение 0
<code>s2, pos</code>	Поиск строки <code>s2</code> , начиная с позиции <code>pos</code> в строке <code>s</code> . По умолчанию <code>pos</code> имеет значение 0
	Поиск строки с завершающим нулевым символом в

cp, pos	стиле С, на которую указывает указатель cp. Поиск начинается с позиции pos в строке s. По умолчанию pos имеет значение 0
cp, pos, n	Поиск первых n символов в массиве, на который указывает указатель cp. Поиск начинается с позиции pos в строке s. Аргумент pos и n не имеют значения по умолчанию



ВНИМАНИЕ

Функции поиска строк возвращают значение беззнакового типа `string::size_type`. Поэтому не следует использовать переменную типа `int` или другого знакового типа для содержания значения, возвращаемого этими функциями (см. раздел 2.1.2).

Самой простой является функция `find()`. Она ищет первое местоположение переданного аргумента и возвращает его индекс или значение `npos`, если соответствующее значение не найдено:

```
string name( "AnnaBelle");
auto pos1 = name.find( "Anna"); // pos1 == 0
```

Возвращает значение 0, т.е. индекс, по которому подстрока "Anna" расположена в строке "AnnaBelle".

Поиск (и другие операции со строками) чувствительны к регистру. При поиске в строке регистр имеет значение:

```
string lowercase( "annabelle");
pos1 = lowercase.find( "Anna"); // pos1 == npos
```

Этот код присвоит переменной pos1 значение `npos`, поскольку строка "Anna" не соответствует строке "anna".

Немного сложней искать соответствие любому символу в строке. Например, следующий код находит первую цифру в переменной name:

```
string numbers( "0123456789"), name( "r2d2");
// возвращает 1, т. е. индекс первой цифры в имени
auto pos = name.find_first_of( numbers);
```

Кроме поиска соответствия, вызывав функцию `find_first_not_of()`, можно искать первую позицию, которая *не соответствует* исковому аргументу. Например, для поиска первого нечислового символа в строке можно использовать следующий код:

```
string dept( "03714p3");
```

```
// возвращает 5 - индекс символа 'p'
auto pos = dept.find_first_not_of(numbers);
```

Откуда начинать поиск

Функциям поиска можно передать необязательный аргумент исходной позиции. Этот необязательный аргумент указывает позицию, с которой начинается поиск. По умолчанию значением этого аргумента является нуль. Общепринятой практикой программирования является использование этого аргумента в цикле перебора строки при поиске всех местоположений искомого значения.

```
string::size_type pos = 0;
// каждая итерация находит следующее число в имени
while ((pos = name.find_first_of(numbers, pos))
       != string::npos) {
    cout << "found number at index: " << pos
        << " element is " << name[pos] << endl;
    ++pos; // перевести на следующий символ
}
```

Условие цикла while присваивает переменной pos индекс первой встретившейся цифры, начиная с текущей позиции pos. Пока функция `find_first_of()` возвращает допустимый индекс, результат отображается, а значение pos увеличивается.

Если не увеличивать значение переменной pos в конце этого цикла, он никогда не завершится, поскольку при последующих итерациях поиск начнется сначала и найден будет тот же элемент. Поскольку значение `npos` так и не будет возвращено, цикл никогда не завершится.

Поиск в обратном направлении

Использованные до сих пор функции поиска выполняются слева направо (т.е. от начала к концу). Библиотека предоставляет аналогичный набор функций, которые просматривают строку справа налево (т.е. от конца к началу). Функция-член `rfind()` ищет последнюю, т.е. расположенную справа, позицию искомой подстроки.

```
string river("Mississippi");
auto first_pos = river.find("is"); // возвращает 1
auto last_pos = river.rfind("is"); // возвращает 4
```

Функция `find()` возвращает индекс 1, указывая, что подстрока "is" первый раз встречается, начиная с позиции 1, а функция `rfind()`

возвращает индекс 4, указывая начало последнего местонахождения подстроки "is".

Функция `find_last()` аналогична функции `find_first()`, но возвращает последнее местоположение, а не первое.

- Функция `find_last_of()` ищет последний символ, который соответствует любому элементу искомой строки.

- Функция `find_last_not_of()` ищет последний символ, который не соответствует ни одному элементу искомой строки.

Каждая из этих функций имеет второй необязательный аргумент, который указывает позицию начала поиска.

Упражнения раздела 9.5.3

Упражнение 9.47. Напишите программу, которая находит в строке "ab2c3d7R4E6" каждую цифру, а затем каждую букву. Напишите две версии программы: с использованием функции `find_first_of()` и функции `find_first_not_of()`.

Упражнение 9.48. С учетом определения переменных `name = "r2d2"` и `numbers = "0123456789"`, что возвращает вызов функции `numbers.find(name)`?

Упражнение 9.49. У символов может быть надстрочная часть, расположенная выше середины строки, как у `d̄` или `f̄`, или подстрочная, ниже середины строки, как у `r̄` или `ḡ`. Напишите программу, которая читает содержащий слова файл и сообщает самое длинное слово, не содержащее ни надстрочных, ни подстрочных элементов.



9.5.4. Сравнение строк

Кроме операторов сравнения (см. раздел 3.2.2), библиотека `string` предоставляет набор функций сравнения, подобных функции `strcmp()` библиотеки С (см. раздел 3.5.4). Подобно функции `strcmp()`, функция `s.compare()` возвращает нуль, положительное или отрицательное значение, в зависимости от того, равна ли строка `s`, больше или меньше строки, переданной ее аргументом.

Как показано в таб. 9.15, существует шесть версий функции `compare()`. Ее аргументы зависят от того, сравниваются ли две строки или строка и символьный массив. В обоих случаях сравнивать можно либо

всю строку, либо ее часть.

Таблица 9.15. Возможные аргументы функции `s. compare()`

<code>s2</code>	Сравнивает строку <code>s</code> со строкой <code>s2</code>
<code>pos1, n1, s2</code>	Сравнивает <code>n1</code> символов, начиная с позиции <code>pos1</code> из строки <code>s</code> , со строкой <code>s2</code>
<code>pos1, n1, s2, pos2, n2</code>	Сравнивает <code>n1</code> символов, начиная с позиции <code>pos1</code> из строки <code>s</code> , со строкой <code>s2</code> , начиная с позиции <code>pos2</code> в строке <code>s2</code>
<code>cp</code>	Сравнивает строку <code>s</code> с завершаемым нулевым символом массивом, на который указывает указатель <code>cp</code>
<code>pos1, n1, cp</code>	Сравнивает <code>n1</code> символов, начиная с позиции <code>pos1</code> из строки <code>s</code> , со строкой <code>cp</code>
<code>pos1, n1, cp, n2</code>	Сравнивает <code>n1</code> символов, начиная с позиции <code>pos1</code> из строки <code>s</code> , со строкой <code>cp</code> , начиная с символа <code>n2</code>



9.5.5. Числовые преобразования

Строки зачастую содержат символы, которые представляют числа. Например, числовое значение 15 можно представить как строку с двумя символами, ' 1 ' и ' 5 ' . На самом деле символьное представление числа отличается от его числового значения. Числовое значение 15, хранимое в 16-разрядной переменной типа `short`, будет иметь двоичное значение 000000000001111, а символьная строка "15", представленная как два символа из набора Latin-1, будет иметь двоичное значение 0011000100110101. Первый байт представляет символ ' 1 ', восьмеричное значение которого составит 061, а второй байт, представляющий символ ' 5 ', в наборе Latin-1 имеет восьмеричное значение 065.



Новый стандарт вводит несколько функций, осуществляющих преобразование между числовыми данными и библиотечным типом `string`.

Таблица 9.16. Преобразования между строками и числами

<code>to_string(val)</code>	Перегруженные версии функции возвращают строковое представление значения <code>val</code> . Аргумент <code>val</code> может иметь любой арифметический тип (см. раздел 2.1.1). Есть версии функции <code>to_string()</code> для любого типа с плавающей точкой и целочисленного типа, включая тип <code>int</code> и большие типы. Малые целочисленные типы преобразуются, как обычно (см. раздел 4.11.1)
<code>stoi(s, p, b)</code> <code>stol(s, p, b)</code> <code>stoul(s, p, b)</code> <code>stoll(s, p, b)</code> <code>stoull(s, p, b)</code>	Возвращают числовое содержимое исходной подстроки <code>s</code> как тип <code>int</code> , <code>long</code> , <code>unsigned long</code> , <code>long long</code> или <code>unsigned long long</code> соответственно. Аргумент <code>b</code> задает используемое для преобразования основание числа; по умолчанию принято значение 10. Аргумент <code>p</code> — указатель на тип <code>size_t</code> , означающий индекс первого нечислового символа в строке <code>s</code> ; по умолчанию <code>p</code> имеет значение 0. В этом случае функция не хранит индекс
<code>stof(s, p)</code> <code>stod(s, p)</code> <code>stold(s, p)</code>	Возвращают числовое содержимое исходной подстроки <code>s</code> как тип <code>float</code> , <code>double</code> или <code>long double</code> соответственно. Аргумент <code>p</code> имеет то же назначение, что и у целочисленных преобразований

```

int i = 42;
string s = to_string(i); // преобразует переменную
i типа int в ее
// символьное
представление
double d = stod(s); // преобразует строку s в
значение типа double

```

Здесь для преобразования числа 42 в его строковое представление используется вызов функции `to_string()`, а затем вызов функции `stod()` преобразует эту строку в значение с плавающей точкой.

Первый преобразуемый в числовое значение символ строки должен быть цифрой:

```

string s2 = "pi = 3.14";
// d = 3.14 преобразуется первая подстрока в строке
s, начинаяющаяся
// с цифры; d = 3.14
d =
stod(s2.substr(s2.find_first_of("+-0123456789")));

```

Для получения позиции первого символа строки `s`, который мог быть частью числа, в этом вызове функции `stod()` используется функция `find_first_of()` (см. раздел 9.5.3). Функции `stod()` передается подстрока строки `s`, начиная с этой позиции. Функция `stod()` читает переданную строку до тех пор, пока не встретится символ, который не может быть частью числа. Затем найденное символьное представление

числа преобразуется в соответствующее значение типа `double`.

Первый преобразуемый символ строки должен быть знаком + или – либо цифрой. Стока может начаться с части 0x или 0X, означающей шестнадцатеричный формат. У функций преобразования чисел с плавающей точкой строка может также начинаться с десятичной точки (.) и содержать символ e или E, означающий экспоненциальную часть. У функций преобразования в целочисленный тип, в зависимости от основания, строка может содержать алфавитные символы, соответствующие цифрам после цифры 9.



Если строка не может быть преобразована в число, эти функции передают исключение `invalid_argument` (см. раздел 5.6). Если преобразование создает значение, которое не может быть представлено заданным типом, они передают исключение `out_of_range`.

Упражнения раздела 9.5.5

Упражнение 9.50. Напишите программу обработки вектора `vector<string>`, элементы которого представляют целочисленные значения. Вычислите сумму всех элементов вектора. Измените программу так, чтобы она суммировала строки, которые представляют значения с плавающей точкой.

Упражнение 9.51. Напишите класс, у которого есть три беззнаковых члена, представляющих год, месяц и день. Напишите конструктор, получающий строку, представляющую дату. Конструктор должен понимать множество форматов даты, такие как January 1, 1900, 1/1/1900, Jan 1, 1900 и т.д.



9.6. Адаптеры контейнеров

Кроме последовательных контейнеров, библиотека предоставляет три адаптера последовательного контейнера: `stack` (стек), `queue` (очередь) и `priority_queue` (приоритетная очередь). *Адаптер* (*adaptor*^[4]) — это фундаментальная концепция библиотеки. Существуют адаптеры

контейнера, итератора и функции. По существу, адаптер — это механизм, заставляющий нечто одно действовать как нечто другое. Адаптер контейнера получает контейнер существующего типа и заставляет его действовать как другой. Например, адаптер `stack` получает любой из последовательных контейнеров (`array` и `forward_list`) и заставляет его работать подобно стеку. Функции и типы данных, общие для всех адаптеров контейнеров, перечислены в табл. 9.17.

Таблица 9.17. Функции и типы, общие для всех адаптеров

<code>size_type</code>	Тип данных, достаточно большой, чтобы содержать размер самого большого объекта этого типа
<code>value_type</code>	Тип элемента
<code>container_type</code>	Тип контейнера, на базе которого реализован адаптер
<code>A a;</code>	Создает новый пустой адаптер по имени <code>a</code>
<code>A a(c);</code>	Создает новый адаптер по имени <code>a</code> , содержащий копию контейнера <code>c</code>
<code>операторы сравнения</code>	Каждый адаптер поддерживает все операторы сравнения: <code>==</code> , <code>!=</code> , <code><</code> , <code><=</code> , <code>></code> и <code>>=</code> . Эти операторы возвращают результат сравнения внутренних контейнеров
<code>a.empty()</code>	Возвращает значение <code>true</code> , если адаптер <code>a</code> пуст, и значение <code>false</code> в противном случае
<code>a.size()</code>	Возвращает количество элементов в адаптере <code>a</code>
<code>swap(a, b)</code> <code>a.swap(b)</code>	Меняет содержимое контейнеров <code>a</code> и <code>b</code> ; у них должен быть одинаковый тип, включая тип контейнера, на основании которого они реализованы

Определение адаптера

Каждый адаптер определяет два конструктора: стандартный конструктор, создающий пустой объект, и конструктор, получающий контейнер и инициализирующий адаптер, копируя полученный контейнер. Предположим, например, что `deq` — это двухсторонняя очередь `deque<int>`. Ее можно использовать для инициализации нового стека следующим образом:

```
stack<int> stk( deq ); // копирует элементы из deq в stk
```

По умолчанию оба адаптера, `stack` и `queue`, реализованы на основании контейнера `deque`, а адаптер `priority_queue` реализован на базе контейнера `vector`. Заданный по умолчанию тип контейнера можно

переопределить, указав последовательный контейнер как второй аргумент при создании адаптера:

```
// пустой стек, реализованный поверх вектора
stack<string, vector<string>> str_stk;
// str_stk2 реализован поверх вектора и
первоначально содержит копию
svec stack<string, vector<string>> str_stk2(svec);
```

Существуют некоторые ограничения на применение контейнеров с определенными адаптерами. Всем адаптерам нужна возможность добавлять и удалять элементы. В результате они не могут быть основаны на массиве. Точно так же нельзя использовать контейнер `forward_list`, поскольку все адаптеры требуют функций добавления, удаления и обращения к последнему элементу контейнера. Стек требует только функций `push_back()`, `pop_back()` и `back()`, поэтому для стека можно использовать контейнер любого из остальных типов. Адаптеру `queue` требуются функции `back()`, `push_back()`, `front()` и `push_front()`, поэтому он может быть создан на основании контейнеров `list` и `deque`, но не `vector`. Адаптеру `priority_queue` в дополнение к функциям `front()`, `push_back()` и `pop_back()` требуется произвольный доступ к элементам; он может быть основан на контейнерах `vector` и `deque`, но не `list`.

Адаптер stack

Тип `stack` определен в заголовке `stack`. Функции-члены класса `stack` перечислены в табл. 9.18. Следующая программа иллюстрирует использование адаптера `stack`:

```
stack<int> intStack; // пустой стек
// заполнить стек
for (size_t ix = 0; ix != 10; ++ix)
    intStack.push(ix); // intStack содержит значения
0...9
while (!intStack.empty()) { // пока в intStack есть
значения
    int value = intStack.top();
    // код, использующий, значение
    intStack.pop(); // извлечь верхний элемент и
повторить
}
```

Сначала `intStack` объявляется как пустой стек целочисленных элементов:

```
stack<int> intStack; // пустой стек
```

Затем цикл `for` добавляет десять элементов, инициализируя каждый следующим целым числом, начиная с нуля. Цикл `while` перебирает весь стек, извлекая его верхний элемент, пока он не опустеет.

Таблица 9.18. Функции, поддерживаемые адаптером контейнера `stack`, кроме приведенных в табл. 9.17

По умолчанию используется контейнер `deque`, но может быть также реализован на основании контейнеров `list` или `vector`.

<code>s.pop()</code>	Удаляет, но не возвращает верхний элемент из стека
<code>s.push(item)</code> <code>s.emplace(args)</code>	Создает в стеке новый верхний элемент, копируя или перемещая элемент <code>item</code> либо создавая элемент из аргумента параметра <code>args</code>
<code>s.top()</code>	Возвращает, но не удаляет верхний элемент из стека

Каждый адаптер контейнера определяет собственные функции, исходя из функций, предоставленных базовым контейнером. Использовать можно только функции адаптера, а функции основного контейнера использовать нельзя. Рассмотрим, например, вызов функции `push_back()` контейнера `deque`, на котором основан стек `intStack`:

```
intStack.push(ix); // intStack содержит значения 0...9
```

Хотя стек реализован на основании контейнера `deque`, прямого доступа к его функциям нет. Для стека нельзя вызвать функцию `push_back()`; вместо нее следует использовать функцию `push()`.

Адаптеры очередей

Адаптеры `queue` и `priority_queue` определены в заголовке `queue`. Список функций, поддерживаемых этими типами, приведен в табл. 9.19.

Таблица 9.19. Функции адаптеров `queue` и `priority_queue`, кроме приведенных в табл. 9.17

По умолчанию адаптер `queue` использует контейнер `deque`, а адаптер `priority_queue` — контейнер `vector`; адаптер `queue` может использовать также контейнер `list` или `vector`, адаптер `priority_queue` может использовать контейнер `deque`.

Удаляет, но не возвращает первый или наиболее приоритетный

q. <code>pop()</code>	элемент из очереди или приоритетной очереди соответственно
q. <code>front()</code> q. <code>back()</code>	Возвращает, но не удаляет первый или последний элемент очереди q. <i>Допустимо только для адаптера queue</i>
q. <code>top()</code>	Возвращает, но не удаляет элемент с самым высоким приоритетом. <i>Допустимо только для адаптера priority_queue</i>
q. <code>push(item)</code> q. <code>emplace(args)</code>	Создает элемент со значением <i>item</i> или создает его исходя из аргумента <i>args</i> в конце очереди или в соответствующей позиции приоритетной очереди

Библиотечный класс `queue` использует хранилище, организованное по принципу "первым пришел, первым вышел" (first-in, first-out — FIFO). Поступающие в очередь объекты помещаются в ее конец, а извлекаются из ее начала.

Адаптер `priority_queue` позволяет установить приоритет хранимых элементов. Добавляемые элементы помещаются перед элементами с более низким приоритетом. По умолчанию для определения относительных приоритетов в библиотеке используется оператор `<`. Его переопределение рассматривается в разделе 11.2.2.

Упражнения раздела 9.6

Упражнение 9.52. Используйте стек для обработки выражений со скобками. Встретив открывающую скобку, запомните ее положение. Встретив закрывающую скобку, после открывающей скобки, извлеките эти элементы, включая открывающую скобку, и поместите полученное значение в стек, переместив таким образом заключенное в скобки выражение.

Резюме

Библиотечные типы контейнеров — это шаблоны, позволяющие содержать объекты указанного типа. В последовательных контейнерах элементы упорядочены и доступны по позиции. У последовательных контейнеров общий, стандартизованный интерфейс: если два последовательных контейнера предоставляют некую функцию, то у нее будет тот же интерфейс и значение в обоих контейнерах.

Все контейнеры (кроме контейнера `array`) обеспечивают эффективное управление динамической памятью. Можно добавлять элементы в контейнер, не волнуясь о том, где хранить элементы. Контейнер сам управляет хранением. Контейнеры `vector` и `string` обеспечивают более

подробное управление памятью, предоставляя функции-члены `reserve()` и `capacity()`.

По большей части, контейнеры определяют удивительно мало функций. Они определяют конструкторы, функции добавления и удаления элементов, функции выяснения размера контейнера и возвращения итераторов на те или иные элементы. Другие весьма полезные функции, такие как сортировка и поиск, определены не типами контейнеров, а стандартными алгоритмами, которые будут описаны в главе 10.

Функции контейнеров, которые добавляют или удаляют элементы, способны сделать существующие итераторы, указатели или ссылки некорректными. Большинство функций, которые способны сделать итераторы недопустимыми, например `insert()` или `erase()`, возвращают новый итератор, позволяющий не потерять позицию в контейнере. Особую осторожность следует соблюдать в циклах, которые используют итераторы и операции с контейнерами, способные изменить их размер.

Термины

Адаптер`priority_queue` (приоритетная очередь). Адаптер последовательных контейнеров, позволяющий создать очередь, в которой элементы добавляются не в конец, а согласно определенному уровню приоритета. По умолчанию при определении приоритета используется оператор "меньше" для типа элемента.

Адаптер`queue` (очередь). Адаптер последовательных контейнеров, позволяющий создать очередь, в которой элементы добавляются в конец, а предоставляются и удаляются в начале.

Адаптер`stack` (стек). Адаптер последовательных контейнеров, позволяющий создать стек, в который элементы добавляют и удаляют только с одного конца.

Адаптер (`adaptor`). Библиотечный тип, функция или итератор, который заставляет один объект действовать подобно другому. Для последовательных контейнеров существуют три адаптера: `stack`, `queue` и `priority_queue`, каждый из которых определяет новый интерфейс базового последовательного контейнера.

Диапазон итераторов (`iterator range`). Диапазон элементов, обозначенный двумя итераторами. Первый итератор относится к первому элементу в последовательности, а второй — к следующему элементу после последнего. Если диапазон пуст, итераторы равны (и наоборот, если

итераторы равны, они обозначают пустой диапазон). Если диапазон не пуст, второй итератор можно достичь последовательным увеличением первого итератора. Последовательное приращение итератора позволяет обработать каждый элемент диапазона.

Интервал, включающий левый элемент (left-inclusive interval). Диапазон значений, включающий первый элемент, но исключающий последний. Обычно обозначается как $[i, j)$, т.е. начальное значение последовательности i включено, а последнее, j , исключено.

Итератор после конца (off-the-end iterator). Итератор, обозначающий (несуществующий) следующий элемент после последнего в диапазоне. Обычно называемый "конечным итератором" (end iterator).

Итератор после начала (off-the-beginning iterator). Итератор, обозначающий (несуществующий) элемент непосредственно перед началом контейнера `forward_list`. Возвращается функцией `before_begin()` контейнера `forward_list`. Подобно итератору, возвращенному функцией `end()`, к значению данного итератора обратиться нельзя.

Контейнер `array` (массив). Последовательный контейнер фиксированного размера. Чтобы определить массив, кроме определения типа элемента следует указать его размер. К элементам массива можно обратиться по их позиционному индексу. Обеспечивает быстрый произвольный доступ к элементам.

Контейнер `deque` (двухсторонняя очередь). Последовательный контейнер, к элементам которого можно обратиться по индексу (позиции). Двухсторонняя очередь подобна вектору во всех отношениях, за исключением того, что он обеспечивает быструю вставку как в начало, так и в конец контейнера, без перемещения элементов.

Контейнер `forward_list` (односвязный список). Последовательный контейнер, представляющий односвязный список. К элементам контейнера `forward_list` можно обращаться только последовательно; начиная с данного элемента, можно добраться до другого элемента, только перебрав каждый элемент между ними. Итераторы класса `forward_list` не поддерживают декремент (`--`). Обеспечивает быструю вставку (или удаление) в любой позиции. В отличие от других контейнеров, вставка и удаление происходят после указанной позиции итератора. Как следствие, у контейнера `forward_list` есть итератор перед началом, для согласованности с обычным итератором после конца. При добавлении новых элементов итераторы остаются допустимыми. Когда элемент

удаляется, некорректным становится лишь итератор удаленного элемента.

Контейнер`list` (список). Последовательный контейнер, к элементам которого можно обратиться только последовательно, т.е. начиная с одного элемента, можно перейти к другому, увеличивая или уменьшая итератор. Итераторы контейнера `list` поддерживают как инкремент (`++`), так и декремент (`--`). Обеспечивает быструю вставку (и удаление) в любой позиции. Добавление новых элементов никак не влияет ни на другие элементы, ни на существующие итераторы. Когда элемент удаляется, некорректным становится лишь итератор удаленного элемента.

Контейнер`vector` (вектор). Последовательный контейнер, к элементам которого можно обратиться по индексу (позиции). Эффективно добавить или удалить элементы вектора можно только с конца. Добавление элементов в вектор может привести к его перераспределению в памяти, что сделает некорректными все созданные ранее итераторы. При добавлении (или удалении) элемента в середину вектора итераторы всех расположенных далее элементов становятся некорректными.

Контейнер (container). Тип (класс), который содержит коллекцию объектов определенного типа. Каждый библиотечный контейнер является шаблоном класса. Чтобы создать контейнер, необходимо указать тип хранимых в нем элементов. За исключением массивов, библиотечные контейнеры имеют переменный размер.

Последовательный контейнер (sequential container). Контейнер, позволяющий содержать упорядоченную коллекцию объектов одинакового типа. К элементам последовательного контейнера обращаются по позиции.

Функция`begin()`. Функция контейнера, возвращающая итератор на первый элемент в контейнере (если он есть), или итератор после конца, если контейнер пуст. Будет ли возвращенный итератор константным, зависит от типа контейнера.

Функция`cbegin()`. Функция контейнера, возвращающая итератор `const_iterator` на первый элемент в контейнере (если он есть), или итератор после конца (off-the-end iterator), если контейнер пуст.

Функция`cend()`. Функция контейнера, возвращающая итератор `const_iterator` на (несуществующий) элемент после последнего элемента контейнера.

Функция`end()`. Функция контейнера, возвращающая итератор на (несуществующий) элемент после последнего элемента контейнера. Будет ли возвращенный итератор константным, зависит от типа контейнера.

Глава 10

Обобщенные алгоритмы

В библиотечных контейнерах определен на удивление небольшой набор функций. Вместо того чтобы снабжать каждый контейнер большим количеством одинаковых функций, библиотека предоставляет набор алгоритмов, большинство из которых не зависит от конкретного типа контейнера. Эти алгоритмы называют *обобщенными* (*generic*), поскольку применимы они и к контейнерам разных типов, и к разным типам элементов.

Темой данной главы являются не только обобщенные алгоритмы, но и более подробное изучение итераторов.

В последовательных контейнерах определено немного операций: по большей части они позволяют добавлять и удалять элементы, обращаться к первому или последнему элементу, определять, не пуст ли контейнер, и получать итераторы на первый элемент или следующий после последнего.

Но может понадобиться и множество других вспомогательных операций: поиск определенного элемента, замена или удаление некого значения, переупорядочивание элементов контейнера и т.д.

Чтобы не создавать каждую из этих функций как член контейнера каждого типа, стандартная библиотека определяет набор *обобщенных алгоритмов* (*generic algorithm*). *Алгоритмами* они называются потому, что реализуют классические алгоритмы, такие как сортировка и поиск, а *обобщенными* — потому, что работают с контейнерами любых типов, включая массивы встроенных типов, и, как будет продемонстрировано далее, с последовательностями других видов, а не только с такими библиотечными типами, как `vector` или `list`.



10.1. Краткий обзор

Большинство алгоритмов определено в заголовке `algorithm`. Библиотека определяет также набор обобщенных числовых алгоритмов в заголовке `numeric`.

Обычно алгоритмы воздействуют не на сам контейнер, а работают, перебирая диапазон элементов, заданный двумя итераторами (см. раздел

9.2.1). Перебирая диапазон, алгоритм, как правило, выполняет некое действие с каждым элементом. Предположим, например, что имеется вектор целых чисел и необходимо узнать, содержит ли этот вектор некое значение. Проще всего ответить на этот вопрос, воспользовавшись библиотечным алгоритмом `find()`:

```
int val = 42; // искомое значение
// result обозначит искомый элемент, если он есть в
// векторе,
// или значение vec.cend(), если нет
auto result = find(vec.cbegin(), vec.cend(), val);
// отображение результата
cout << "The value " << val
     << (result == vec.cend()
         ? " is not present" : " is present") <<
endl;
```

Первые два аргумента функции `find()` являются итераторами, обозначающими диапазон элементов, а третий аргумент — это значение. Функция `find()` сравнивает каждый элемент указанного диапазона с заданным значением и возвращает итератор на первый элемент, соответствующий этому значению. При отсутствии соответствия функция `find()` возвращает свой второй итератор, означая неудачу поиска. Так, сравнив возвращаемое значение со вторым аргументом, можно определить, был ли найден элемент. Для проверки, свидетельствующей об успехе поиска значения, в операторе вывода используется условный оператор (см. раздел 4.7).

Поскольку функция `find()` работает с итераторами, ее можно использовать для поиска значения в любом контейнере. Рассмотрим пример применения функции `find()` для поиска значения в списке строк:

```
string val = "a value"; // искомое значение
// вызов, позволяющий найти строковый элемент в
// списке
auto result = find(lst.cbegin(), lst.cend(), val);
```

Аналогично, поскольку указатели действуют как итераторы встроенных массивов, функцию `find()` можно использовать для поиска в массиве:

```
int ia[] = { 27, 210, 12, 47, 109, 83 };
int val = 83;
```

```
int* result = find(begin(ia), end(ia), val);
```

Здесь для передачи указателей на первый и следующий после

последнего элементы массива `ia` используются библиотечные функции `begin()` и `end()` (см. раздел 3.5.3).

Искать также можно в диапазоне, заданном переданными итераторами (или указателями), на его первый и следующий после последнего элементы. Например, следующий вызов ищет соответствие в элементах `ia[1]`, `ia[2]` и `ia[3]`:

```
// искать среди элементов, начиная с ia[ 1 ] и до, но
// не включая, ia[ 4 ]
auto result = find( ia +1, ia + 4, val);
```

Как работают алгоритмы

Для того чтобы изучить применение алгоритмов к контейнерам различных типов, немного подробней рассмотрим функцию `find()`. Ее задачей является поиск указанного элемента в не отсортированной коллекции элементов. Концептуально функция `find()` должна выполнить следующие действия.

1. Обратиться к первому элементу последовательности.
2. Сравнить этот элемент с искомым значением.
3. Если элемент соответствует искомому, функция `find()` возвращает значение, идентифицирующее этот элемент.
4. В противном случае функция `find()` переходит к следующему элементу и повторяет этапы 2 и 3.
5. По достижении конца последовательности функция `find()` должна остановиться.
6. Достигнув конца последовательности, функция `find()` должна возвратить значение, означающее неудачу поиска. Тип этого значения должен быть совместимым с типом значения, возвращенного на этапе 3.

Ни одно из этих действий не зависит от типа контейнера, который содержит элементы. Пока есть итераторы, применимые для доступа к элементам, функция `find()` в любом случае не зависит от типа контейнера (или даже хранимых в контейнере элементов).

Итераторы делают алгоритмы независимыми от типа контейнера...

Все этапы работы функции `find()`, кроме второго, могут быть выполнены средствами итератора: оператор обращения к значению итератора предоставляет доступ к значению элемента; если элемент соответствует искомому, функция `find()` может возвратить итератор на

этот элемент; оператор инкремента итератора переводит его на следующий элемент; итератор после конца будет означать достижение функцией `find()` конца последовательности; функция `find()` вполне может возвратить итератор после конца (см. раздел 9.2.1), чтобы указать на неудачу поиска.

...но алгоритмы зависят от типа элементов

Хотя итераторы делают алгоритмы независимыми от контейнеров, большинство алгоритмов используют одну (или больше) функцию типа элемента. Например, этап 2 использует оператор `==` типа элемента для сравнения каждого элемента с предоставленным значением.

Другие алгоритмы требуют, чтобы тип элемента имел оператор `<`. Но, как будет продемонстрировано, большинство алгоритмов позволяют предоставить собственную функцию для использования вместо оператора, заданного по умолчанию.

Упражнения раздела 10.1

Упражнение 10.1. В заголовке `algorithm` определена функция `count()`, подобная функции `find()`. Она получает два итератора и значение, а возвращает количество обнаруженных в диапазоне элементов, обладающих искомым значением. Организуйте чтение в вектор последовательности целых чисел. Осуществите подсчет элементов с указанным значением.

Упражнение 10.2. Повторите предыдущую программу, но чтение значений организуйте в список (`list`) строк.

Ключевая концепция. Алгоритмы никогда не используют функции контейнеров

Общие алгоритмы никогда не используют функции контейнеров. Они работают исключительно с итераторами. Тот факт, что алгоритмы оперируют итераторами, а не функциями контейнера, возможно, удивителен, но он имеет глубокий смысл: когда используются "обычные" итераторы, алгоритмы не способны изменить размер исходного контейнера. Как будет продемонстрировано далее, алгоритмы способны изменять значения хранимых в контейнере элементов и перемещать их в контейнер. Однако они не способны ни добавлять, ни удалять сами элементы.

Как будет продемонстрировано в разделе 10.4.1, существуют специальные классы итераторов, которые способны на несколько

большее, чем просто перебор элементов. Они позволяют выполнять операции вставки. Когда алгоритм работает с одним из таких *операторов*, возможен побочный эффект добавления элемента в контейнер, однако в самих *алгоритмах* это никогда не используется.



10.2. Первый взгляд на алгоритмы

Библиотека предоставляет больше ста алгоритмов. К счастью, у алгоритмов, как и у контейнеров, единообразная архитектура. Понимание этой архитектуры упростит изучение и использование всех ста с лишним алгоритмов без необходимости изучать каждый из них. В этой главе рассматривается использование алгоритмов и описаны характеризующие их общие принципы. В приложении А перечислены все алгоритмы согласно принципам их работы.

За небольшим исключением, все алгоритмы работают с диапазоном элементов. Далее этот диапазон мы будем называть *исходным диапазоном* (*input range*). Алгоритмы, работающие с исходным диапазоном, всегда получают его в виде двух первых параметров. Эти параметры являются итераторами, используемыми для обозначения первого и следующего после последнего элемента, подлежащих обработке.

Несмотря на то что большинство алгоритмов работают с одинаково обозначенным исходным диапазоном, они отличаются тем, как используются элементы этого диапазона. Проще всего подразделить алгоритмы на читающие, записывающие и меняющие порядок элементов.



10.2.1. Алгоритмы только для чтения

Много алгоритмов только читают значения элементов в исходном диапазоне, но никогда не записывают их. Функция `find()` и функция `count()`, использованная в упражнениях раздела 10.1, являются примерами таких алгоритмов.

Другим предназначенным только для чтения алгоритмом является `accumulate()`, который определен в заголовке `numeric`. Функция `accumulate()` получает три аргумента. Первые два определяют диапазон суммируемых элементов, а третий — исходное значение для суммы. Предположим, что `vec` — это последовательность целых чисел.

```
// суммирует элементы вектора vec, начиная со
значения 0
```

```
int sum = accumulate( vec.cbegin(), vec.cend(), 0);
```

Приведенный выше код суммирует значения элементов вектора `vec`, используя 0 как начальное значение суммы.



Тип третьего аргумента функции `accumulate()` определяет, какой именно оператор суммы будет использован и каков будет тип возвращаемого значения функции `accumulate()`.

Алгоритмы и типы элементов

У того факта, что функция `accumulate()` использует свой третий аргумент как отправную точку для суммирования, есть важное последствие: он позволяет добавить тип элемента к типу суммы. Таким образом, тип элементов последовательности должен соответствовать или быть приводим к типу третьего аргумента. В этом примере элементами вектора `vec` могли бы быть целые числа, или числа типа `double`, или `long long`, или любого другого типа, который может быть добавлен к значению типа `int`.

Например, поскольку тип `string` имеет оператор `+`, функцию `accumulate()` можно использовать для конкатенации элементов вектора строк:

```
string sum = accumulate( v.cbegin(), v.cend(), string("") );
```

Этот вызов добавляет каждый элемент вектора `v` к первоначально пустой строке `sum`. Обратите внимание: третий параметр здесь явно указан как объект класса `string`. Передача строки как символьного литерала привела бы к ошибке при компиляции.

```
// ошибка: no + on const char*
```

```
string sum = accumulate( v.cbegin(), v.cend(), "");
```

Если бы был передан строковый литерал, типом суммируемых значений оказался бы `const char*`. Этот тип и определяет используемый оператор `+`. Поскольку тип `const char*` не имеет оператора `+`, этот вызов не будет компилироваться.

С алгоритмами, которые читают, но не пишут в элементы, обычно лучше использовать функции `cbegin()` и `cend()` (см. раздел 9.2.3). Но если возвращенный алгоритмом итератор планируется использовать для изменения значения элемента, то следует использовать функции `begin()` и `end()`.



Алгоритмы, работающие с двумя последовательностями

Еще один алгоритм только для чтения, `equal()`, позволяет определять, содержит ли две последовательности одинаковые значения. Он сравнивает каждый элемент первой последовательности с соответствующим элементом второй. Алгоритм возвращает значение `true`, если соответствующие элементы равны, и значение `false` в противном случае. Он получает три итератора: первые два (как обычно) обозначают диапазон элементов первой последовательности, а третий — первый элемент второй последовательности:

```
// roster2 должен иметь по крайней мере столько же
элементов,
// сколько и roster1
equal(roster1.cbegin(), roster1.cend(),
roster2.cbegin());
```

Поскольку функция `equal()` работает с итераторами, ее можно вызвать для сравнения элементов контейнеров разных типов. Кроме того, типы элементов также не обязаны совпадать, пока можно использовать оператор `==` для их сравнения. Например, контейнер `roster1` мог быть вектором `vector<string>`, а контейнер `roster2` — списком `list<const char*>`.

Однако алгоритм `equal()` делает критически важное предположение: подразумевает, что вторая последовательность по крайней мере не меньше первой. Этот алгоритм просматривает каждый элемент первой последовательности и подразумевает, что для него есть соответствующий элемент во второй последовательности.



Алгоритмы, получающие один итератор, обозначающий вторую

последовательность, подразумевают, что вторая последовательность по крайней мере не меньше первой.

Упражнения раздела 10.2.1

Упражнение 10.3. Примените функцию `accumulate()` для суммирования элементов вектора `vector<int>`.

Упражнение 10.4. Если вектор `v` имеет тип `vector<double>`, в чем состоит ошибка вызова `accumulate(v.cbegin(), v.cend(), 0)` (если она есть)?

Упражнение 10.5. Что произойдет, если в вызове функции `equal()` для списков оба из них будут содержать строки в стиле С, а не библиотечные строки?

Ключевая концепция. Итераторы, передаваемые в качестве аргументов

Некоторые алгоритмы читают элементы из двух последовательностей. Составляющие эти последовательности элементы могут храниться в контейнерах различных видов. Например, первая последовательность могла бы быть вектором (`vector`), а вторая списком (`list`), двухсторонней очередью (`deque`), встроенным массивом или другой последовательностью. Кроме того, типы элементов этих последовательностей не обязаны совпадать точно. Обязательно необходима возможность сравнивать элементы этих двух последовательностей. Например, в алгоритме `equal()` типы элемента не обязаны быть идентичными, на самом деле нужна возможность использовать оператор `==` для сравнения элементов этих двух последовательностей.

Алгоритмы, работающие с двумя последовательностями, отличаются способом передачи второй последовательности. Некоторые алгоритмы, такие как `equal()`, получают три итератора: первые два обозначают диапазон первой последовательности, а третий — обозначает первый элемент во второй последовательности. Другие получают четыре итератора: первые два обозначают диапазон элементов в первой последовательности, а вторые два — диапазон второй последовательности.

Алгоритмы, использующие для обозначения второй последовательности один итератор, подразумевают, что вторая последовательность по крайней мере не меньше первой. Разработчик

должен сам позаботиться о том, чтобы алгоритм не пытался обратиться к несуществующим элементам во второй последовательности. Например, алгоритм `equal()` сравнивает каждый элемент первой последовательности с соответствующим элементом второй. Если вторая последовательность является подмножеством первой, то возникает серьезная ошибка — функция `equal()` попытается обратиться к элементам после конца второй последовательности.



10.2.2. Алгоритмы, записывающие элементы контейнера

Некоторые алгоритмы способны записывать значения в элементы. Используя такие алгоритмы, следует соблюдать осторожность и предварительно удостовериться, что количество элементов, в которые алгоритм собирается внести изменения, по крайней мере не превосходит количество существующих элементов. Помните, что алгоритмы работают не с контейнерами, поэтому у них нет возможности самостоятельно изменить размер контейнера.

Некоторые алгоритмы осуществляют запись непосредственно в исходную последовательность. Эти алгоритмы в принципе безопасны: они способны переписать лишь столько элементов, сколько находится в указанном исходном диапазоне.

В качестве примера рассмотрим алгоритм `fill()`, получающий два итератора, обозначающих диапазон, и третий аргумент, являющийся значением. Функция `fill()` присваивает данное значение каждому элементу исходной последовательности:

```
fill( vec.begin(), vec.end(), 0); // обнулить каждый элемент
```

```
// присвоить половине последовательности значение 10
```

```
fill( vec.begin(), vec.begin() + vec.size()/2, 10);
```

Поскольку функция `fill()` пишет в переданную ей исходную последовательность до тех пор, пока она не закончится, запись вполне безопасна.



Алгоритмы не проверяют операции записи

Некоторые алгоритмы получают итератор, обозначающий конкретное назначение. Эти алгоритмы присваивают новые значения элементам последовательности, начиная с элемента, обозначенного итератором назначения. Например, функция `fill_n()` получает один итератор, количество и значение. Она присваивает предоставленное значение заданному количеству элементов, начиная с элемента, обозначенного итератором. Функцию `fill_n()` можно использовать для присвоения нового значения элементам вектора:

```
vector<int> vec; // пустой вектор
// используя вектор vec, предоставить ему разные
значения
fill_n(vec.begin(), vec.size(), 0); // обнулить
каждый элемент vec
```

Функция `fill_n()` подразумевала, что безопасно запишет указанное количество элементов. Таким образом, следующий вызов функции `fill_n()` подразумевает, что `dest` указывает на существующий элемент и что в последовательности есть по крайней мере `n` элементов, начиная с элемента `dest`.

```
fill_n(dest, n, val)
```

Это вполне обычная ошибка для новичка: вызов функции `fill_n()` (или подобного алгоритма записи элементов) для контейнера без элементов:

```
vector<int> vec; // пустой вектор
// катастрофа: попытка записи в 10 несуществующих
элементов
// вектора vec
fill_n(vec.begin(), 10, 0);
```

Этот вызов функции `fill_n()` ведет к катастрофе. Должно быть записано десять элементов, но вектор `vec` пуст, и никаких элементов в нем нет. Результат непредсказуем, но, вероятнее всего, произойдет серьезный отказ во время выполнения.



Алгоритмы, осуществляющие запись по итератору назначения, подразумевают, что контейнер достаточно велик для содержания всех

записываемых элементов.

Функция *back_inserter*()

Один из способов проверки, имеет ли контейнер достаточно элементов для записи, подразумевает использование *итератора вставки* (*insert iterator*), который позволяет *добавлять* элементы в базовый контейнер. Как правило, при присвоении значения элементу контейнера при помощи итератора осуществляется присвоение тому элементу, на который указывает итератор. При присвоении с использованием итератора вставки в контейнер добавляется новый элемент, равный правому значению.

Более подробная информация об итераторе вставки приведена в разделе 10.4.1. Однако для иллюстрации безопасного применения алгоритмов, записывающих данные в контейнер, используем функцию *back_inserter()*, определенную в заголовке *iterator*.

Функция *back_inserter()* получает ссылку на контейнер и возвращает итератор вставки, связанный с данным контейнером. Попытка присвоения значения элементу при помощи этого итератора приводит к вызову функции *push_back()*, добавляющей элемент с данным значением в контейнер.

```
vector<int> vec; // пустой вектор
auto it = back_inserter(vec); // присвоение при
помощи it добавляет
                                         // элементы в vec
*it = 42; // теперь vec содержит один элемент со
значением 42
```

Функцию *back_inserter()* зачастую применяют для создания итератора, используемого в качестве итератора назначения алгоритмов. Рассмотрим пример:

```
vector<int> vec; // пустой вектор
// ok: функция back_inserter() создает итератор
вставки,
// который добавляет элементы в вектор vec
fill_n(back_inserter(vec), 10, 0); // добавляет 10
элементов в vec
```

На каждой итерации функция *fill_n()* присваивает элемент заданной последовательности. Поскольку ей передается итератор, возвращенный функцией *back_inserter()*, каждое присвоение вызовет функцию *push_back()* вектора *vec*. В результате этот вызов функции

`fill_n()` добавит в конец вектора `vec` десять элементов, каждый со значением 0.

Алгоритм `copy()`

Алгоритм `copy()` — это еще один пример алгоритма, записывающего элементы последовательности вывода, обозначенной итератором назначения. Этот алгоритм получает три итератора. Первые два обозначают исходный диапазон, а третий — начало последовательности вывода. Этот алгоритм копирует элементы из исходного диапазона в элементы вывода. Важно, чтобы переданный функции `copy()` контейнер вывода был не меньше исходного диапазона.

В качестве примера используем функцию `copy()` для копирования одного встроенного массива в другой:

```
int a1[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int a2[ sizeof(a1) / sizeof(*a1) ]; // a2 имеет тот же
размер, что и a1
// указывает на следующий элемент после последнего
скопированного в a2
auto ret = copy( begin(a1), end(a1), a2 ); // копирует a1 в a2
```

Здесь определяется массив по имени `a2`, а функция `sizeof()` используется для гарантии равенства размеров массивов `a2` и `a1` (см. раздел 4.9). Затем происходит вызов функции `copy()` для копирования массива `a1` в массив `a2`. После вызова у элементов обоих массивов будут одинаковые значения.

Возвращенное функцией `copy()` значение является приращенным значением ее итератора назначения. Таким образом, итератор `ret` укажет на следующий элемент после последнего скопированного в массив `a2`.

Некоторые алгоритмы обладают так называемой версией "копирования". Эти алгоритмы осуществляют некую обработку элементов исходной последовательности, но саму последовательность не изменяют. Они могут создавать новую последовательность, в которую и сохраняют результат обработки элементов исходной.

Например, алгоритм `replace()` читает последовательность и заменяет каждый экземпляр заданного значения другим значением. Алгоритм получает четыре параметра: два итератора, обозначающих исходный диапазон, и два значения. Он заменяет вторым значением значение каждого элемента, которое равно первому.

```
// заменить во всех элементах значение 0 на 42  
replace(ilist.begin(), ilist.end(), 0, 42);
```

Этот вызов заменяет все экземпляры со значением 0 на 42. Если исходную последовательность следует оставить неизменной, необходимо применить алгоритм `replace_copy()`. Этой версии функции передают третий аргумент: итератор, указывающий получателя откорректированной последовательности.

```
// использовать функцию back_inserter() для  
увеличения контейнера  
// назначения до необходимых размеров  
replace_copy(ilist.cbegin(), ilist.cend(),  
            back_inserter(vec), 0, 42);
```

После этого вызова список `ilist` останется неизменным, а вектор `vec` будет содержать копию его элементов, но со значением 42 вместо 0.



10.2.3. Алгоритмы, переупорядочивающие элементы контейнера

Некоторые алгоритмы изменяют порядок элементов в пределах контейнера. Яркий пример такого алгоритма — `sort()`. Вызов функции `sort()` упорядочивает элементы исходного диапазона в порядке сортировки, используя оператор `<` типа элемента.

Предположим, необходимо проанализировать слова, использованные в наборе детских рассказов. Текст рассказов содержится в векторе. Необходимо сократить этот вектор так, чтобы каждое слово присутствовало в нем только один раз, независимо от того, сколько раз оно встречается в любом из данных рассказов.

Для иллюстрации поставленной задачи используем в качестве исходного текста следующую простую историю:

the quick red fox jumps over the slow red turtle

В результате обработки этого текста программа должна создать следующий вектор:

fox	jumps	over	quick	red	red	slow	the	the	turtle
-----	-------	------	-------	-----	-----	------	-----	-----	--------

Устранение дубликатов

Для устранения повторяющихся слов сначала отсортируем вектор так,

чтобы дубликаты располагались рядом друг с другом. После сортировки вектора можно использовать другой библиотечный алгоритм, `unique()`, чтобы расположить уникальные элементы в передней части вектора. Поскольку алгоритмы не могут работать с самими контейнерами, используем функцию-член `erase()` класса `vector` для фактического удаления элементов:

```
void elimDups( vector<string> &words) {  
    // сортировка слов в алфавитном порядке позволяет  
    // найти дубликаты  
    sort( words.begin(), words.end() );  
    // функция unique( ) переупорядочивает исходный  
    // диапазон так, чтобы  
    // каждое слово присутствовало только один раз в  
    // начальной части  
    // диапазона, и возвращает итератор на элемент,  
    // следующий после  
    // диапазона уникальных значений  
    auto end_unique = unique( words.begin(),  
    words.end() );  
    // для удаления не уникальных элементов используем  
    // функцию erase( ) вектора  
    words.erase( end_unique, words.end() );  
}
```

Алгоритм `sort()` получает два итератора, обозначающих диапазон элементов для сортировки. В данном случае сортируется весь вектор. После вызова функции `sort()` слова упорядочиваются так:

fox	jumps	over	quick	red	red	slow	the	the	turtle
-----	-------	------	-------	-----	-----	------	-----	-----	--------

Обратите внимание: слова `red` и `the` встречаются дважды.



Алгоритм `unique()`

После сортировки слов необходимо оставить только один экземпляр каждого из них. Алгоритм `unique()` перестраивает исходный диапазон так, чтобы устраниТЬ смежные повторяющиеся элементы, и возвращает итератор, обозначающий конец диапазона уникальных значений. После вызова функции `unique()` вектор выглядит так:



Размер вектора `words` не изменился: в нем все еще десять элементов. Изменился только порядок этих элементов — смежные дубликаты были как бы "удалены". Слово *удалены* заключено в кавычки потому, что функция `unique()` не удаляет элементы. Она переупорядочивает смежные дубликаты так, чтобы уникальные элементы располагались в начале последовательности. Возвращенный функцией `unique()` итератор указывает на следующий элемент после последнего уникального. Последующие элементы все еще существуют, но их значение уже не важно.



Библиотечные алгоритмы работают с итераторами, а не с контейнерами. Поэтому алгоритм не может непосредственно добавить или удалить элементы.



Применение функций контейнера для удаления элементов

Для фактического удаления неиспользуемых элементов следует использовать контейнерную функцию `erase()` (см. раздел 9.3.3). Удалению подлежит диапазон элементов от того, на который указывает итератор `end_unique`, и до конца контейнера `words`. После вызова контейнер `words` содержит восемь уникальных слов из исходного текста.

Следует заметить, что вызов функции `erase()` окажется безопасным, даже если вектор не содержит совпадающих слов. В этом случае функция `unique()` возвратит итератор, совпадающий с возвращенным функцией `word.end()`. Таким образом, оба аргумента функции `erase()` будут иметь одинаковое значение, а следовательно, обрабатываемый ею диапазон окажется пустым. Удаление пустого диапазона не приводит ни к какому результату, поэтому программа будет работать правильно даже тогда, когда в исходном тексте нет повторяющихся слов.

Упражнения раздела 10.2.3

Упражнение 10.6. Напишите программу, использующую функцию `fill_n()` для обнуления последовательности целых чисел.

Упражнение 10.7. Определите, есть ли ошибки в следующих фрагментах кода, и, если есть, как их исправить:

- (a) `vector<int> vec; list<int> lst; int i;`
`while (cin >> i)`
`lst.push_back(i);`
`copy(lst.cbegin(), lst.cend(), vec.begin());`
- (b) `vector<int> vec;`
`vec.reserve(10); // reserve рассматривается в`
разделе 9.4
`fill_n(vec.begin(), 10, 0);`

Упражнение 10.8. Как упоминалось, алгоритмы не изменяют размер контейнеров, с которыми они работают. Почему использование функции `back_inserter()` не противоречит этому утверждению?

Упражнение 10.9. Реализуйте собственную версию функции `elimDups()`. Проверьте ее в программе, выводящей содержимое вектора после чтения ввода, после вызова функции `unique()` и после вызова функции `erase()`.

Упражнение 10.10. Почему алгоритмы не изменяют размер контейнеров?

10.3. Перенастройка функций

Большинство алгоритмов сравнивает элементы исходной последовательности. По умолчанию такие алгоритмы используют оператор `<` или `==` типа элемента. Библиотека предоставляет также версии этих алгоритмов, позволяющие использовать собственный оператор вместо заданного по умолчанию.

Например, алгоритм `sort()` использует оператор `<` типа элемента. Но может понадобиться сортировать последовательность в порядке, отличном от определенного оператором `<`, либо у типа элемента последовательности может не быть оператора `<` (как у класса `Sales_data`). В обоих случаях необходимо переопределить стандартное поведение функции `sort()`.



10.3.1. Передача функций алгоритму

Предположим, например, что необходимо вывести вектор после вызова функции `elimDups()` (см. раздел 10.2.3). Однако слова должны быть упорядочены сначала по размеру, а затем в алфавитном порядке в пределах каждого размера. Чтобы переупорядочить вектор по длине слов, используем вторую перегруженную версию функции `sort()`. Она получает третий аргумент, называемый предикатом.

Предикаты

Предикат (*predicate*) — это допускающее вызов выражение, возвращающее значение, применимое в условии. Библиотечные алгоритмы используют *унарные предикаты* (*unary predicate*) (с одним параметром) или *бинарные предикаты* (*binary predicate*) (с двумя параметрами). Получающие предикаты алгоритмы вызывают его для каждого элемента в исходном диапазоне. Поэтому тип элемента должен допускать преобразование в тип параметра предиката.

Версия функции `sort()`, получающей бинарный предикат, использует его вместо оператора `<` при сравнении элементов. Предикаты, предоставляемые функции `sort()`, должны соответствовать требованиям, описанным в разделе 11.2.2, а пока достаточно знать, что он должен определить единообразный порядок для всех возможных элементов в

исходной последовательности. Функция `isShorter()` из раздела 6.2.2 — хороший пример функции, соответствующей этим требованиям. Таким образом, функцию `isShorter()` можно передать как предикат алгоритму `sort()`. Это переупорядочит элементы по размеру:

```
// функция сравнения, используемая при сортировке
// слов по длине
bool isShorter( const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
// сортировка слов по длине от коротких к длинным
sort( words.begin(), words.end(), isShorter);
```

Если контейнер `words` содержит те же данные, что и в разделе 10.2.3, то этот вызов переупорядочит его так, что все слова длиной 3 символа расположатся перед словами длиной 4 символа, которые в свою очередь расположатся перед словами длиной 5 символов, и т.д.

Алгоритмы сортировки

При сортировке вектора `words` по размеру следует также обеспечить алфавитный порядок элементов одинаковой длины. Для обеспечения алфавитного порядка можно использовать алгоритм `stable_sort()`, обеспечивающий первоначальный порядок сортировки среди равных элементов.

Обычно об относительном порядке равных элементов можно не заботиться. В конце концов, они ведь равны. Но в данном случае под "равными" подразумеваются элементы со значениями одинаковой длины. Элементы одинаковой длины все еще отличаются друг от друга при просмотре их содержимого. Вызов функции `stable_sort()` позволяет расположить элементы с одинаковыми значениями длины в алфавитном порядке:

```
elimDups( words); // расположить слова в алфавитном
// порядке
// и удалить дубликаты
// пересортировать по длине, поддерживая алфавитный
// порядок среди слов
// той же длины
stable_sort( words.begin(), words.end(), isShorter);
for (const auto &s : words) // копировать строки не
```

НУЖНО

```
cout << s << " "; // вывести каждый элемент,  
отделяя его пробелом  
cout << endl;
```

Предположим, что перед этим вызовом вектор words был отсортирован в алфавитном порядке. После вызова он будет отсортирован по размеру элемента, а слова одинаковой длины остаются в алфавитном порядке. Если выполнить этот код для первоначального содержимого вектора, то результат будет таким:

```
fox red the over slow jumps quick turtle
```

Упражнения раздела 10.3.1

Упражнение 10.11. Напишите программу, использующую функции `stable_sort()` и `isShorter()` для сортировки вектора, переданного вашей версии функции `elimDups()`. Для проверки правильности программы выведите содержимое вектора.

Упражнение 10.12. Напишите функцию `compareIsbn()`, которая сравнивает члены `isbn` двух объектов класса `Sales_data`. Используйте эту функцию для сортировки вектора объектов класса `Sales_data`.

Упражнение 10.13. Библиотека определяет алгоритм `partition()`, получающий предикат и делящий контейнер так, чтобы значения, для которых предикат возвращает значение `true`, располагались в начале последовательности, а для которых он возвращает значение `false` — в конце. Алгоритм возвращает итератор на следующий элемент после последнего, для которого предикат возвратил значение `true`. Напишите функцию, которая получает строку и возвращает логическое значение, указывающее, содержит ли строка пять символов или больше. Используйте эту функцию для разделения вектора `words`. Выведите элементы, у которых есть пять или более символов.

10.3.2. Лямбда-выражения

У передаваемых алгоритмам предикатов должен быть точно один или два параметра, в зависимости от того, использует ли алгоритм унарный или бинарный предикат соответственно. Но иногда необходима обработка, которая требует большего количества аргументов, чем позволяет предикат алгоритма. Например, решение для последнего упражнения в предыдущем разделе имело жестко заданный размер в 5 символов, согласно которому предикат делил последовательность. Было бы удобней иметь возможность разделять последовательность без необходимости писать отдельный предикат для каждого возможного размера.

Для примера пересмотрим программу из раздела 10.3.1 так, чтобы вывести количество слов с указанным размером или больше него. Вывод изменим так, чтобы он сообщал только те слова, длина которых равна или больше заданной.

Вот "эскиз" этой функции, которую мы назовем `biggies()`:

```
void biggies( vector<string> &words,
vector<string>::size_type sz) {
    elimDups( words); // расположить слова в алфавитном
// порядке
                                // и удалить дубликаты
    // пересортировать по длине, поддерживая
алфавитный порядок среди слов
    // той же длины
        stable_sort( words.begin(), words.end(),
isShorter);
    // получить итератор на первый элемент, размер
которого >= sz
    // вычислить количество элементов с размером >= sz
    // вывести слова, размер которых равен или больше
заданного, разделяя
    // их пробелами
}
```

Новая проблема — поиск первого элемента вектора заданного размера. Как известно, чтобы выяснить количество элементов, размер которых равен или больше заданного, можно использовать их позицию.

Для поиска элементов определенного размера можно использовать библиотечный алгоритм `find_if()`. Подобно функции `find()` (см.

раздел 10.1), алгоритм `find_if()` получает два итератора, обозначающих диапазон. В отличие от функции `find()`, третий аргумент функции `find_if()` является предикатом. Алгоритм `find_if()` вызывает переданный предикат для каждого элемента в исходном диапазоне. Он возвращает первый элемент, для которого предикат возвращает отличное от нуля значение, или конечный итератор, если ни один подходящий элемент не найден.

Совсем не сложно написать функцию, которая получает строку и размер, а возвращает логическое значение, означающее, не превосходит ли размер данной строки указанный. Однако функция `find_if()` получает унарный предикат, поэтому любая передаваемая ей функция, которая может быть вызвана с элементом исходной последовательности, должна иметь только один параметр. Нет никакого способа передать ей второй аргумент, представляющий размер. Чтобы решить эту часть проблемы, используем некоторые дополнительные средства языка.

Знакомство с лямбда-выражением

Алгоритму можно передать любой вид *вызываемого объекта* (callable object). Объект или выражение является вызываемым, если к нему можно применить оператор вызова (см. раздел 1.5.2). Таким образом, если `e` — вызываемое выражение, то можно написать `e(аргументы)`, где *аргументы* — это разделяемый запятыми список любого количества аргументов.

Единственными вызываемыми объектами, использованными до сих пор, были функции и указатели на функции (см. раздел 6.7). Есть еще два вида вызываемых объектов: классы, перегружающие оператор вызова функции (будут рассматриваться в разделе 14.8), и *лямбда-выражения* (lambda expression).



Лямбда-выражение представляет собой вызываемый блок кода. Его можно считать безымянной встраиваемой функцией. Подобно любой функции, у лямбда-выражений есть тип возвращаемого значения, список параметров и тело функции. В отличие от функции, лямбда-выражения могут быть определены в функции. Форма лямбда-выражений такова:

```
[ список захвата ] ( список параметров ) -> тип
возвращаемого значения
{ тело функции }
```

где список захвата (зачастую пустой) — это список локальных переменных, определенных в содержащей функции; тип возвращаемого значения, список параметров и тело функции — те же самые, что и у любой обычной функции. Однако, в отличие от обычных функций, для определения типа возвращаемого значения лямбда-выражения должны использовать замыкающий тип (см. раздел 6.3.3).

Список параметров и типа возвращаемого значения могут отсутствовать, но список захвата и тело функции должны быть всегда:

```
auto f = [] { return 42; };
```

Здесь `f` определено как вызываемый объект, не получающий никаких аргументов и возвращающий значение 42. Вызов лямбда-выражений происходит таким же способом, что и вызов функций, — при помощи оператора вызова:

```
cout << f() << endl; // выводит 42
```

Пропуск круглых скобок и списка параметров в лямбда-выражении эквивалентен определению пустого списка параметров. Следовательно, когда происходит вызов лямбда-выражения `f`, список аргументов оказывается пустым. Если пропущен тип возвращаемого значения, то выведенный тип возвращаемого значения лямбда-выражения будет зависеть от кода в теле функции. Если телом является только оператор `return`, тип возвращаемого значения выводится из типа возвращаемого выражения. В противном случае типом возвращаемого значения является `void`.



Лямбда-выражения, тела которых содержат нечто кроме одного оператора `return`, что не определяет тип возвращаемого значения, возвращают тип `void`.

Передача аргументов лямбда-выражению

Подобно вызовам обычных функций, аргументы вызова лямбда-выражения используются для инициализации его параметров. Как обычно, типы аргумента и параметра должны совпадать. В отличие от обычных функций, у лямбда-выражений не может быть аргументов по умолчанию (см. раздел 6.5.1). Поэтому у вызова лямбда-выражения всегда

аргументов, сколько и параметров. Как только параметры инициализируются, выполняется тело лямбда-выражения.

Для примера передачи аргументов можно написать лямбда-выражение, ведущее себя как функция `isShorter()`:

```
[ ](const string &a, const string &b)
{ return a.size() < b.size();}
```

Пустой список захвата означает, что это лямбда-выражение не будет использовать локальных переменных из окружающей функции. Параметры лямбда-выражения, как и параметры функции `isShorter()`, будут ссылкой на константную строку. Как и тело функции `isShorter()`, тело лямбда-выражения сравнивает размер параметров и возвращает логическое значение, зависящее от соотношения размеров своих аргументов.

Вызов функции `stable_sort()` можно переписать так, чтобы использовать это лямбда-выражение следующим образом:

```
// сортировать слова по размеру, поддерживая
алфавитный порядок среди
// слов того же размера
stable_sort(words.begin(), words.end(),
            [ ](const string &a, const string &b)
            { return a.size() < b.size();});
```

Когда функция `stable_sort()` будет сравнивать два элемента, она вызовет данное лямбда-выражение.

Использование списка захвата

Теперь все готово для решения первоначальной задачи — создания вызываемого выражения, которое можно передать функции `find_if()`. Необходимо выражение, сравнивающее длину каждой строки в исходной последовательности со значением параметра `sz` функции `biggies()`.

Хотя лямбда-выражение может присутствовать в функции, оно способно использовать локальные переменные этой функции, только заранее определив, какие из них предстоит использовать. Лямбда-выражение определяет подлежащие использованию локальные переменные, включив их в *список захвата* (*capture list*). Список захвата предписывает лямбда-выражению включить информацию, необходимую для доступа к этим переменным, в само лямбда-выражение.

В данном случае лямбда-выражение захватит переменную `sz` и будет иметь один параметр типа `string`. Тело лямбда-выражения сравнивает размер переданной строки с захваченным значением переменной `sz`:

```
[ sz]( const string &a)
{ return a.size () >= sz; };
```

В начинающих лямбда-выражение квадратных скобках, [], можно расположить разделяемый запятыми список имен, определенных в окружающей функции.

Поскольку данное лямбда-выражение захватывает переменную sz, ее можно будет использовать в теле лямбда-выражения. Лямбда-выражение не захватывает вектор words, поэтому доступа к его переменным она не имеет. Если бы лямбда-выражение имело пустой список захвата, наш код не компилировался бы:

```
// ошибка: sz не захвачена
[ ]( const string &a)
{ return a.size() >= sz; };
```



Лямбда-выражение может использовать локальную переменную окружающей функции, только если она присутствует в ее списке захвата.

Вызов функции `find_if()`

Используя это лямбда-выражение, можно найти первый элемент, размер которого не меньше sz:

```
// получить итератор на первый элемент, размер
которого >= sz
auto wc = find_if(words.begin(), words.end(),
                   [ sz]( const string &a)
                   { return a.size() >= sz; } );
```

Вызов функции `find_if()` возвращает итератор на первый элемент, длина которого не меньше sz, или на элемент `words.end()`, если такого элемента не существует.

Возвращенный функцией `find_if()` итератор можно использовать для вычисления количества элементов, расположенных между этим итератором и концом вектора words (см. раздел 3.4.2):

```
// вычислить количество элементов с размером >= sz
auto count = words.end() - wc;
cout << count << " " << make_plural(count, "word",
"s")
```

```
<< " of length " << sz << " or longer" <<  
endl;
```

Для вывода в сообщении слова `word` или `words`, в зависимости от того, равен ли размер 1, оператор вывода вызывает функцию `make_plural()` (см. раздел 6.3.2).

Алгоритм `for_each()`

Последняя часть задачи — вывод тех элементов вектора `words`, длина которых не меньше `sz`. Для этого используем алгоритм `for_each()`, получающий вызываемый объект и вызывающий его для каждого элемента в исходном диапазоне:

```
// вывести слова, размер которых равен или больше  
зданного, разделяя  
// их пробелами  
for_each(wc, words.end(),  
         [](const string &s) { cout << s << " ";});  
cout << endl;
```

Список захвата этого лямбда-выражения пуст, но все же его тело использует два имени: его собственный параметр `s` и `cout`.

Список захвата пуст, поскольку он используется только для нестатических переменных, определенных в окружающей функции. Лямбда-выражение вполне может использовать имена, определенные вне той функции, в которой присутствует лямбда-выражение. В данном случае имя `cout` не локально определено в функции `biggies()`, оно определено в заголовке `iostream`. Пока заголовок `iostream` находится в области видимости функции `biggies()`, данное лямбда-выражение может использовать имя `cout`.



Список захвата используется только для локальных нестатических переменных; лямбда-выражения могут непосредственно использовать статические локальные переменные и переменные, объявленные вне функции.

Объединив все вместе

Теперь, изучив элементы программы подробно, рассмотрим ее в целом:

```

void biggies( vector<string> &words,
              vector<string>::size_type sz) {
    elimDups( words); // расположить слова в алфавитном
порядке
                                // и удалить дубликаты
    // пересортировать по длине, поддерживая
алфавитный порядок среди слов
    // той же длины
    stable_sort( words.begin(), words.end(),
                  []( const string &a, const string &b)
                  { return a.size() < b.size(); });
    // получить итератор на первый элемент, размер
которого >= sz
    auto wc = find_if( words.begin(), words.end(),
                      [sz]( const string &a)
                      { return a.size() >= sz; });
    // вычислить количество элементов с размером >= sz
    auto count = words.end() - wc;
    cout << count << " " << make_plural( count, "word",
"s")
                                << " of length " << sz << " or longer" <<
endl;
    // вывести слова, размер которых равен или больше
зданного, разделяя
    // их пробелами
    for_each( wc, words.end(),
              [] (const string &s) { cout << s << " "; });
    cout << endl;
}

```

Упражнения раздела 10.3.2

Упражнение 10.14. Напишите лямбда-выражение, получающее два целых числа и возвращающее их сумму.

Упражнение 10.15. Напишите лямбда-выражение, захватывающее переменную типа `int` окружающей функции и получающей параметр типа `int`. Лямбда-выражение должно возвратить сумму захваченного значения типа `int` и параметра типа `int`.

Упражнение 10.16. Напишите собственную версию функции `biggies()`, используя лямбда-выражения.

Упражнение 10.17. Перепишите упражнение 10.12 из раздела 10.3.1 так, чтобы в вызове функции `sort()` вместо функции `compareIsbn()` использовалось лямбда-выражение.

Упражнение 10.18. Перепишите функцию `biggies()` так, чтобы использовать алгоритм `partition()` вместо алгоритма `find_if()`. Алгоритм `partition()` описан в упражнении 10.13 раздела 10.3.1.

Упражнение 10.19. Перепишите предыдущее упражнение так, чтобы использовать алгоритм `stable_partition()`, который, подобно алгоритму `stable_sort()`, обеспечивает исходный порядок элементов в разделяемой последовательности.

10.3.3. Захват и возвращение значений лямбда-выражениями

При определении лямбда-выражения компилятор создает новый (безымянный) класс, соответствующий этому лямбда-выражению. Создание этих классов рассматривается в разделе 14.8.1, а пока следует понять, что при передаче лямбда-выражения функции определяется новый тип и создается его объект. Безымянnyй объект этого созданного компилятором типа и передается как аргумент. Аналогично при использовании ключевого слова `auto` для определения переменной, инициализированной лямбда-выражением, определяется объект типа, созданного из этого лямбда-выражения.

По умолчанию созданный из лямбда-выражения класс содержит переменные-члены, соответствующие захваченным переменным лямбда-выражения. Подобно переменным-членам любого класса, переменные-члены лямбда-выражения инициализируются при создании его объекта.

Захват по значению

Подобно передаче параметров, переменные можно захватывать по значению или по ссылке. В табл. 10.1 приведены различные способы создания списка захвата. До сих пор у использованных лямбда-выражений захват переменных осуществлялся по значению. Подобно передаче по значению параметров, захват переменной по значению подразумевает ее копирование. Но, в отличие от параметров, копирование значения при захвате осуществляется при создании лямбда-выражения, а не при его вызове:

```
void fcn1() {
    size_t v1 = 42; // локальная переменная
    // копирует v1 в вызываемый объект f
    auto f = [v1] { return v1; };
    v1 = 0;
    auto j = f(); // j = 42; f получит копию v1 на
    момент создания
}
```

Поскольку значение копируется при создании лямбда-выражения, последующие изменения захваченной переменной никак не влияют на соответствующее значение в лямбда-выражении.

Таблица 10.1. Список захвата лямбда-выражения

[]	Пустой список захвата. Лямбда-выражение не может использовать переменные из содержащей функции. Лямбда-выражение может использовать локальные переменные, только если оно захватывает их
[<i>names</i>]	<i>names</i> — разделяемый запятыми список имен, локальных для содержащей функции. По умолчанию переменные в списке захвата копируются. Имя, которому предшествует знак &, захватывается по ссылке
[&]	Неявный захват по ссылке. Сущности из содержащей функции используются в теле лямбда-выражения по ссылке
[=]	Неявный захват по значению. Сущности из содержащей функции используются в теле лямбда-выражения как копии
[&, <i>identifier_list</i>]	<i>identifier_list</i> — разделяемый запятыми список любого количества переменных из содержащей функции. Эти переменные захватываются по значению; любые неявно захваченные переменные захватываются по ссылке. Именам в списке <i>identifier_list</i> не могут предшествовать символы &
[=, <i>reference_list</i>]	Переменные, включенные в список <i>reference_list</i> , захватываются по ссылке; любые неявно захваченные переменные захватываются по значению. Имена в списке <i>reference_list</i> не могут включать часть <i>this</i> и должны предваряться символом &

Захват по ссылке

Можно также определять лямбда-выражения, захватывающие переменные по ссылке. Например:

```
void fcn2() {
    size_t v1 = 42; // локальная переменная
    // объект f2 содержит ссылку на v1
    auto f2 = [ &v1] { return v1; };
    v1 = 0;
    auto j = f2(); // j = 0; f2 ссылается на v1; он не
хранится в j
}
```

Символ & перед v1 указывает, что переменная v1 должна быть захвачена как ссылка. Захваченная по ссылке переменная действует так же, как любая другая ссылка. При использовании переменной в теле лямбда-выражения фактически применяется объект, с которым связана эта ссылка. В данном случае, когда лямбда-выражение возвращает v1, возвращается значение объекта, на который ссылается переменная v1.

Захват ссылок имеет те же проблемы и ограничения, что и возвращение ссылок (см. раздел 6.1.1). При захвате переменной по ссылке следует быть *уверенным*, что объект, на который она ссылается, существует на момент выполнения лямбда-выражения. Переменные, захваченные лямбда-выражением, являются локальными, они перестают существовать сразу, как только функция завершится. Если лямбда-выражение продолжит выполняться после завершения функции, то используемые ею локальные переменные окажутся несуществующими.

Иногда захват по ссылке необходим. Например, может понадобиться, чтобы функция `biggies()` получала ссылку на поток `ostream` для записи символа, используемого как разделитель:

```
void biggies( vector<string> &words,
              vector<string>::size_type sz,
              ostream &os = cout, char c = ' ' ) {
    // код, переупорядочивающий слова как прежде
    // оператор вывода количества, пересмотренный для
    // вывода os
    for_each( words.begin(), words.end(),
              [ &os, c]( const string &s) { os << s << c;
} );
}
```

Объекты потока `ostream` нельзя копировать (см. раздел 8.1.1); единственный способ захвата объекта `os` — это ссылка (или указатель).

При передаче лямбда-выражения функции, как и в случае вызова функции `for_each()`, лямбда-выражение выполняется немедленно. Захват объекта `os` по ссылке хорош потому, что переменные в функции `biggies()` существуют во время выполнения функции `for_each()`.

Лямбда-выражение можно также возвратить из функции. Функция может возвратить вызываемый объект непосредственно или возвратить объект класса, у которого вызываемый объект является переменной-членом. Если функция возвращает лямбда-выражение, то по тем же причинам, по которым функция не должна возвращать ссылку на локальную переменную, лямбда-выражение не должно содержать захваченных ссылок.



Когда переменная захвачена по ссылке, следует удостовериться, что эта

переменная существует во время выполнения лямбда-выражения.

Совет. Не усложняйте списки захвата лямбда-выражений

Механизм захвата лямбда-выражения хранит полученную информацию между моментом создания лямбда-выражение (т.е. когда выполняется код определения лямбда-выражения) и моментом собственно выполнения лямбда-выражения. Разработчику следует самостоятельно позаботиться о том, чтобы независимо от момента захвата информации она осталась достоверной на момент выполнения лямбда-выражения.

Захват обычной переменной (типа `int`, `string` и так далее, но не указателя) обычно достаточно прост. В данном случае следует позаботиться о наличии у переменной значения в момент ее захвата.

При захвате указателя, итератора или переменной по ссылке следует удостовериться, что связанный с ними объект все еще существует на момент *выполнения* лямбда-выражения. Кроме того, объект в этот момент гарантированно должен иметь значение. Код, выполняемый между моментом создания лямбда-выражения и моментом его выполнения, может изменить значение объекта, на который указывает (или ссылается) захваченная сущность. Во время захвата указателя (или ссылки) значение объекта, возможно, и было правильным, но ко времени выполнения лямбда-выражения оно могло измениться.

Как правило, сокращая объем захватываемых данных, потенциальных проблем с захватом можно избежать. Кроме того, по возможности избегайте захвата указателей и ссылок.

Неявный захват

Вместо предоставления явного списка переменных содержащей функции, которые предстоит использовать, можно позволить компилятору самостоятельно вывести используемые переменные из кода тела лямбда-выражения. Чтобы заставить компилятор самостоятельно вывести список захвата, в нем используется символ `&` или `=`. Символ `&` указывает, что предполагается захват по ссылке, а символ `=` — что значения захватываются по значению. Например, передаваемое функции `find_if()` лямбда-выражение можно переписать так:

```
// sz неявно захватывается по значению
wc = find_if(words.begin(), words.end(),
              [=](const string &s)
```

```
{ return s.size () >= sz; } );
```

Если одни переменные необходимо захватить по значению, а другие по ссылке, вполне можно совместить явный и неявный захваты:

```
void biggies( vector<string> &words,
               vector<string>::size_type sz,
               ostream &os = cout, char c = ' ' ) {
    // другие действия, как прежде
    // os неявно захватывается по ссылке;
    // с явно захватывается по значению
    for_each( words.begin(), words.end(),
              [ &, c]( const string &s) { os << s << c;
} );
    // os явно захватывается по ссылке;
    // с неявно захватывается по значению
    for_each( words.begin(), words.end(),
              [=, &os]( const string &s) { os << s << c;
} );
}
```

При совмещении неявного и явного захвата первым элементом в списке захвата должен быть символ `&` или `=`. Эти символы задают режим захвата по умолчанию: по ссылке или по значению соответственно.

При совмещении неявного и явного захвата явно захваченные переменные должны использовать дополнительную форму. Таким образом, при неявном захвате по ссылке (с использованием `&`) явно именованные переменные должны захватываться по значению; следовательно, их именам не может предшествовать символ `&`. И наоборот, при неявном захвате по значению (с использованием `=`) явно именованным переменным должен предшествовать символ `&`, означающий, что они должны быть захвачены по ссылке.

Изменяемые лямбда-выражения

По умолчанию лямбда-выражение не может изменить значение переменной, которую она копирует по значению. Чтобы изменить значение захваченной переменной, за списком параметров должно следовать ключевое слово `mutable`. Изменяемые лямбда-выражения не могут пропускать список параметров:

```
void fcn3() {
    size_t v1 = 42; // локальная переменная
    // f может изменить значение захваченных
```

переменных

```
auto f = [ v1]() mutable { return ++v1; };
v1 = 0;
auto j = f(); // j = 43
}
```

Может ли захваченная по ссылке переменная быть изменена, зависит только от того, ссылается ли она на константный или неконстантный тип:

```
void fcn4() {
    size_t v1 = 42; // локальная переменная
    // v1 - ссылка на неконстантную переменную
    // эту переменную можно изменить в f2 при помощи
    // ссылки
    auto f2 = [ &v1] { return ++v1; };
    v1 = 0;
    auto j = f2(); // j = 1
}
```

Определение типа возвращаемого значения лямбда-выражения

Использованные до сих пор лямбда-выражения содержали только один оператор `return`. В результате тип возвращаемого значения определять было не нужно. По умолчанию, если тело лямбда-выражения содержало какие-нибудь операторы, кроме оператора `return`, то подразумевалось, что оно возвращало тип `void`. Подобно другим функциям, возвращающим тип `void`, подобные лямбда-выражения могут не возвращать значения.

В качестве примера используем библиотечный алгоритм `transform()` и лямбда-выражение для замены каждого отрицательного значения в последовательности его абсолютным значением:

```
transform( vi.begin(), vi.end(), vi.begin(),
           [](int i) { return i < 0 ? -i : i; } );
```

Функция `transform()` получает три итератора и вызываемый объект. Первые два итератора обозначают исходную последовательность, третий итератор — назначение. Алгоритм вызывает переданный ему вызываемый объект для каждого элемента исходной последовательности и записывает результат по назначению. Как и в данном примере, итератор назначения может быть тем же, обозначающим начало ввода. Когда исходный итератор и итератор назначения совпадают, алгоритм `transform()` заменяет каждый элемент в исходном диапазоне результатом вызова вызываемого объекта для этого элемента.

В этом вызове передавалось лямбда-выражение, которое возвращает абсолютное значение своего параметра. Тело лямбда-выражения — один оператор `return`, который возвращает результат условного выражения. Необходимости определять тип возвращаемого значения нет, поскольку его можно вывести из типа условного оператора.

Но если написать на первый взгляд эквивалентную программу, используя оператор `if`, то код не будет компилироваться:

```
// ошибка: нельзя вывести тип возвращаемого
значения лямбда-выражения
transform( vi.begin(), vi.end(), vi.begin(),
           [](int i) { if (i < 0) return -i; else
return i; } );
```

Эта версия лямбда-выражения выводит тип возвращаемого значения как `void`, но возвращает значение.



Когда необходимо определить тип возвращаемого значения для лямбда-выражения, следует использовать замыкающий тип возвращаемого значения (см. раздел 6.3.3):

```
transform( vi.begin(), vi.end(), vi.begin(),
           [](int i) -> int
             { if (i < 0) return -i; else return i;
} );
```

В данном случае четвертым аргументом функции `transform()` является лямбда-выражение с пустым списком захвата, единственным параметром типа `int` и возвращаемым значением типа `int`. Его телом является оператор `if`, возвращающий абсолютное значение параметра.

Упражнения раздела 10.3.3

Упражнение 10.20. Библиотека определяет алгоритм `count_if()`. Подобно алгоритму `find_if()`, он получает пару итераторов, обозначающих исходный диапазон и предикат, применяемый к каждому элементу заданного диапазона. Функция `count_if()` возвращает количество раз, когда предикат вернул значение `true`. Используйте алгоритм `count_if()`, чтобы переписать ту часть программы, которая рассчитывала количество слов длиной больше 6.

Упражнение 10.21. Напишите лямбда-выражение, которое захватывает локальную переменную типа `int` и осуществляет декремент ее значения,

пока оно не достигает 0. Как только значение переменной достигнет 0, декремент переменной прекращается. Лямбда-выражение должно возвратить логическое значение, указывающее, имеет ли захваченная переменная значение 0.



10.3.4. Привязка аргументов

Лямбда-выражения особенно полезны для простых операций, которые не предполагается использовать в более чем одном или двух местах. Если ту же операцию необходимо осуществлять во многих местах, то обычно определяют функцию, а не повторяют то же лямбда-выражение многократно. Аналогично, если операция требует многих операторов, обычно лучше использовать функцию.

Как правило, вместо лямбда-выражения с пустым списком захвата проще использовать функцию. Как уже упоминалось, для упорядочивания вектора по длине слов можно использовать или лямбда-выражение, или нашу функцию `isShorter()`. Точно так же совсем не сложно заменить лямбда-выражение, выводившее содержимое вектора, функцией, которая получает строку и выводит ее на стандартное устройство вывода.

Однако не так просто написать функцию для замены лямбда-выражений, которые захватывают локальные переменные. Рассмотрим, например, использованное в вызове функции `find_if()` лямбда-выражение, которое сравнивало размер строки с заданным размером. Совсем не сложно написать функцию, выполняющую те же действия:

```
bool check_size(const string &s, string::size_type sz) {
    return s.size() >= sz;
}
```

Но мы не можем использовать эту функцию как аргумент функции `find_if()`. Как уже упоминалось, функция `find_if()` получает унарный предикат, поэтому переданное ей вызываемое выражение должно получать один аргумент. Лямбда-выражение, переданное функцией `biggies()` функции `find_if()`, использует свой список захвата для хранения значения переменной `sz`. Чтобы использовать функцию `check_size()` вместо этого лямбда-выражения, следует выяснить, как передать аргумент `sz` параметру.

Библиотечная функция `bind()`



Проблему передачи аргумента размера функции `check_size()` можно

решить при помощи новой библиотечной функции `bind()`, определенной в заголовке `functional`. Функцию `bind()` можно считать универсальным адаптером функции (см. раздел 9.6). Она получает вызываемый объект и создает новый вызываемый объект, который адаптирует список параметров исходного объекта.

Общая форма вызова функции `bind()` такова:

```
auto новыйВызываемыйОбъект = bind( вызываемыйОбъект,  
список_аргументов);
```

где `новыйВызываемыйОбъект` — это новый вызываемый объект, а `список_аргументов` — разделяемый запятыми список аргументов, соответствующих параметрам переданного вызываемого объекта `вызываемыйОбъект`. Таким образом, когда происходит вызов объекта `новыйВызываемыйОбъект`, он вызывает `вызываемыйОбъект`, передавая аргументы из списка `список_аргументов`.

Аргументы из списка `список_аргументов` могут включать имена в формате `_n`, где `n` — целое число. Эти аргументы — знакоместа, представляющие параметры объекта `новыйВызываемыйОбъект`. Они располагаются *вместо* аргументов, которые будут переданы объекту `новыйВызываемыйОбъект`. Число `n` является позицией параметра вновь созданного вызываемого объекта: `_1` — первый параметр, `_2` — второй и т.д.

Привязка параметра sz к функции check_size()

В качестве примера использования функции `bind()` создадим объект, который вызывает функцию `check_size()` с фиксированным значением ее параметра размера:

```
// check6 - вызываемый объект, получающий один  
аргумент типа string  
// изывающий функцию check_size() с этой строкой  
и значением 6
```

```
auto check6 = bind(check_size, _1, 6);
```

У этого вызова функции `bind()` есть только одно знакоместо, означающее, что вызываемый объект `check6()` получает один аргумент. Знакоместо располагается первым в списке аргументов. Это означает, что параметр вызываемого объекта `check6()` соответствует первому параметру функции `check_size()`. Этот параметр имеет тип `const string&`, а значит, параметр вызываемого объекта `check6()` также

имеет тип `const string&`. Таким образом, при вызове `check6()` следует передать аргумент типа `string`, который вызываемый объект `check6()` передаст в качестве первого аргумента функции `check_size()`.

Второй аргумент в списке аргументов (т.е. третий аргумент функции `bind()`) является значением 6. Это значение связывается со вторым параметром функции `check_size()`. Каждый раз, когда происходит вызов вызываемого объекта `check6()`, он передает функции `check_size()` значение 6 как второй аргумент:

```
string s = "hello";
bool b1 = check6(s); // check6(s) вызывает
check_size(s, 6)
```

Используя функцию `bind()`, можно заменить следующий исходный вызов функции `find_if()` на базе лямбда-выражения:

```
auto wc = find_if(words.begin(), words.end(),
                   [sz](const string &a)
```

кодом, использующим функцию `check_size()`,

```
auto wc = find_if(words.begin(), words.end(),
                   bind(check_size, _1, sz));
```

Этот вызов функции `bind()` создает вызываемый объект, который привязывает второй аргумент функции `check_size()` к значению параметра `sz`. Когда функция `find_if()` вызовет этот объект для строк вектора `words`, он, в свою очередь, вызовет функцию `check_size()`, передав полученную строку и значение параметра `sz`. Таким образом, функция `find_if()` фактически вызовет функцию `check_size()` для каждой строки в исходном диапазоне и сравнит размер этой строки со значением параметра `sz`.

Использование имен из пространства имен `placeholders`

Имена формата `_n` определяются в пространстве имен `placeholders`. Само это пространство имен определяется в пространстве имен `std` (см. раздел 3.1). Чтобы использовать эти имена, следует предоставить имена обоих пространств имен. Подобно нашим другим примерам, данные вызовы функции `bind()` подразумевали наличие соответствующих объявлений `using`. Рассмотрим объявление `using` для имени `_1`:

```
using std::placeholders::_1;
```

Это объявление свидетельствует о том, что используется имя `_1`,

определенное в пространстве имен `placeholders`, которое само определено в пространстве имен `std`.

Для каждого используемого имени знакомства следует предоставить отдельное объявление `using`. Но поскольку написание таких объявлений может быть утомительно и ведет к ошибкам, вместо этого можно использовать другую форму `using`, которая подробно рассматривается в разделе 18.2.2:

```
using namespace пространствоимен_имя;
```

Она свидетельствует, что необходимо сделать доступными для нашей программы все имена из пространства имен `пространствоимен_имя`:

```
using namespace std::placeholders;
```

Этот код позволяет использовать все имена, определенные в пространстве имен `placeholders`. Подобно функции `bind()`, пространство имен `placeholders` определено в заголовке `functional`.

Аргументы функции `bind()`

Как уже упоминалось, функцию `bind()` можно использовать для фиксированного значения параметра. В более общем случае функцию `bind()` можно использовать для привязки или перестройки параметров в предоставленном вызываемом объекте. Предположим, например, что `f()` — вызываемый объект с пятью параметрами:

```
// g - вызываемый объект, получающий два аргумента
auto g = bind(f, a, b, _2, c, _1);
```

Этот вызов функции `bind()` создает новый вызываемый объект, получающий два аргумента, представленные знакомствами `_2` и `_1`. Новый вызываемый объект передает собственные аргументы как третий и пятый аргументы вызываемому объекту `f()`. Первый, второй и четвертый аргументы вызываемого объекта `f()` связаны с переданными значениями `a`, `b` и `c` соответственно.

Аргументы вызываемого объекта `g()` связаны со знакомствами по позиции. Таким образом, первый аргумент вызываемого объекта `g()` связан с параметром `_1`, а второй — с параметром `_2`. Следовательно, когда происходит вызов `g()`, его первый аргумент будет передан как последний аргумент вызываемого объекта `f()`; второй аргумент `g()` будет передан как третий. В действительности этот вызов функции `bind()` преобразует вызов `g(_1, _2)` в вызов `f(a, b, _2, c, _1)`.

Таким образом, вызов вызываемого объекта `g()` вызывает вызываемый объект `f()` с использованием аргументов вызываемого объекта `g()` для

знакомест наряду с аргументами a, b и c. Например, вызов g(X, Y) приводит к вызову f(a, b, Y, c, X) .

Использование функции bind() для переупорядочивания параметров

Рассмотрим более конкретный пример применения функции bind() для переупорядочивания аргументов. Используем ее для обращения смысла функции isShorter() следующим образом:

```
// сортировка по длине слов от коротких к длинным
sort(words.begin(), words.end(), isShorter);
// сортировка по длине слов от длинных к коротким
sort(words.begin(), words.end(), bind(isShorter,
_2, _1));
```

В первом вызове, когда алгоритм sort() должен сравнить два элемента, A и B, он вызовет функцию isShorter(A, B) . Во втором вызове аргументы функции isShorter() меняются местами. В данном случае, когда алгоритм sort() сравнивает элементы, он вызывает функцию isShorter(B, A) .

Привязка ссылочных параметров

По умолчанию аргументы функции bind() , не являющиеся знакоместами, копируются в возвращаемый ею вызываемый объект. Однако, подобно лямбда-выражениям, иногда необходимо связать аргументы, которые следует передать по ссылке, или необходимо связать аргумент, тип которого не допускает копирования.

Для примера заменим лямбда-выражение, которое захватывало поток ostream по ссылке:

```
// os - локальная переменная, ссылающаяся на поток
// вывода
// c - локальная переменная типа char
for_each(words.begin(), words.end(),
[&os, c] (const string &s) { os << s << c;
});
```

Вполне можно написать функцию, выполняющую ту же задачу:

```
ostream &print(ostream &os, const string &s, char
c) {
    return os << s << c;
}
```

Но для замены захвата переменной `os` нельзя использовать функцию `bind()` непосредственно:

```
// ошибка: нельзя копировать os
for_each(words.begin(), words.end(), bind(print,
os, _1, ' '));
```

Поскольку функция `bind()` копирует свои аргументы, она не сможет скопировать поток `ostream`. Если объект необходимо передать функции `bind()`, не копируя, то следует использовать библиотечную функцию `ref()`:

```
for_each(words.begin(), words.end(),
bind(print, ref(os), _1, ' '));
```

Функция `ref()` возвращает объект, который содержит переданную ссылку, являясь при этом вполне копируемым. Существует также функция `cref()`, создающая класс, содержащий ссылку на константу. Подобно функции `bind()`, функции `ref()` и `cref()` определены в заголовке `functional`.

Совместимость с прежней версией: привязка аргументов

Прежние версии языка C++ имели много больше ограничений, и все же более сложный набор средств привязки аргументов к функциям. Библиотека определяет две функции — `bind1st()` и `bind2nd()`. Подобно функции `bind()`, эти функции получают функцию и создают новый вызываемый объект для вызова переданной функции с одним из ее параметров, связанным с переданным значением. Но эти функции могут связать только первый или второй параметр соответственно. Поскольку они имеют очень много ограничений, в новом стандарте эти функции *не рекомендованы*. Это устаревшее средство, которое может не поддерживаться в будущих выпусках. Современные программы C++ должны использовать функцию `bind()`.

Упражнения раздела 10.3.4

Упражнение 10.22. Перепишите программу подсчета слов размером 6 символов с использованием функций вместо лямбда-выражений.

Упражнение 10.23. Сколько аргументов получает функции `bind()`?

Упражнение 10.24. Используйте функции `bind()` и `check_size()` для поиска первого элемента вектора целых чисел, значение которого больше длины заданного строкового значения.

Упражнение 10.25. В упражнениях раздела 10.3.2 была написана

версия функции `biggies()`, использующая алгоритм `partition()`. Перепишите эту функцию так, чтобы использовать функции `check_size()` и `bind()`.

10.4. Возвращаясь к итераторам

В дополнение к итераторам, определяемым для каждого из контейнеров, библиотека определяет в заголовке `iterator` несколько дополнительных видов итераторов.

- *Итератор вставки* (`insert iterator`). Связан с контейнером и применяется для вставки элементов в контейнер.
- *Потоковый итератор* (`stream iterator`). Может быть связан с потоком ввода или вывода и применяется для перебора связанного потока ввода-вывода.
- *Реверсивный итератор* (`reverse iterator`). Перемещается назад, а не вперед. У всех библиотечных контейнеров, кроме `forward_list`, есть реверсивные итераторы.
- *Итератор перемещения* (`move iterator`). Итератор специального назначения; он перемещает элементы, а не копирует. Эти итераторы рассматриваются в разделе 13.6.2.



10.4.1. Итераторы вставки

Адаптер вставки (`inserter`), или адаптер `inserter`, — это адаптер итератора (см. раздел 9.6), получающий контейнер и возвращающий итератор, позволяющий вставлять элементы в указанный контейнер. Присвоение значения при помощи итератора вставки приводит к вызову контейнерной функции, добавляющей элемент в определенную позицию заданного контейнера. Операторы, поддерживающие эти итераторы, приведены в табл. 10.2.

Таблица 10.2. Операторы итератора вставки

<code>it = t</code>	Вставляет значение <code>t</code> в позицию, обозначенную итератором <code>it</code> . В зависимости от вида итератора вставки и с учетом того, что он связан с контейнером <code>c</code> , вызывает функции <code>c.push_back(t)</code> , <code>c.push_front(t)</code> и <code>c.insert(t, p)</code> , где <code>p</code> — позиция итератора, заданная адаптеру вставки
<code>*it, ++it, it++</code>	Эти операторы существуют, но ничего не делают с итератором <code>it</code> . Каждый оператор возвращает итератор <code>it</code>

Существуют три вида адаптеров вставки, которые отличаются позицией

добавляемых элементов.

- Адаптер `back_inserter` (см. раздел 10.2.2) создает итератор, использующий функцию `push_back()`.
- Адаптер `front_inserter` создает итератор, использующий функцию `push_front()`.
- Адаптер `inserter` создает итератор, использующий функцию `insert()`. Кроме имени контейнера, адаптеру `inserter` передают второй аргумент: итератор, указывающий позицию, перед которой должна начаться вставка.



Адаптер `front_inserter` можно использовать, *только* если у контейнера есть функция `push_front()`. Аналогично адаптер `back_inserter` можно использовать, *только* если у контейнера есть функция `push_back()`.

Важно понимать, что при вызове адаптера `inserter(c, iter)` возвращается итератор, который при использовании вставляет элементы перед элементом, первоначально обозначенным итератором `iter`. Таким образом, если `it` — итератор, созданный адаптером `inserter`, то присвоение `*it = val;` ведет себя, как следующий код:

```
it = c.insert(it, val); // it указывает на недавно
добавленный элемент
++it; // инкремент it, чтобы он указывал на тот же
элемент,
// что и прежде
```

Итератор, созданный адаптером `front_inserter`, ведет себя прямо противоположно итератору, созданному адаптером `inserter`. При использовании адаптера `front_inserter` элементы всегда вставляются перед текущим первым элементом контейнера. Даже если переданная адаптеру `inserter` позиция первоначально обозначает первый элемент, то, как только будет вставлен элемент перед этим элементом, он больше не будет элементом в начале контейнера:

```
list<int> lst = {1, 2, 3, 4};
list<int> lst2, lst3; // пустой список
// после завершения копирования, lst2 содержит 4 3
```

```
copy( lst.cbegin(), lst.cend(),  
      front_inserter(lst2));  
      // после завершения копирования, lst3 содержит 1 2  
3 4  
copy( lst.cbegin(), lst.cend(), inserter(lst3,  
lst3.begin()));
```

Когда происходит вызов `front_inserter(c)`, возвращается итератор вставки, который последовательно вызывает функцию `push_front()`. По мере вставки каждого элемента он становится новым первым элементом контейнера `c`. Следовательно, адаптер `front_inserter` возвращает итератор, который полностью изменяет порядок последовательности, в которую осуществляется вставка; адаптеры `inserter` и `back_inserter` так не поступают.

Упражнения раздела 10.4.1

Упражнение 10.26. Объясните различия между тремя итераторами вставки.

Упражнение 10.27. В дополнение к функции `unique()` (см. раздел 10.2.3) библиотека определяет функцию `unique_copy()`, получающую третий итератор, обозначающий назначение копирования уникальных элементов. Напишите программу, которая использует функцию `unique_copy()` для копирования уникальных элементов вектора в первоначально пустой список.

Упражнение 10.28. Скопируйте вектор, содержащий значения от 1 до 9, в три других контейнера. Используйте адаптеры `inserter`, `back_inserter` и `front_inserter` соответственно для добавления элементов в эти контейнеры. Предскажите вид результирующей последовательности в зависимости от вида адаптера вставки и проверьте свои предсказания на написанной программе.



10.4.2. Потоковые итераторы

Хотя типы `iostream` не относятся к контейнерам, есть итераторы, применимые к объектам типов ввода-вывода (см. раздел 8.1). Итератор `istream_iterator` (табл. 10.3) читает входной поток, а итератор `ostream_iterator` (табл. 10.4) пишет в поток вывода. Эти итераторы рассматривают свой поток как последовательность элементов определенного типа. Используя потоковый итератор, можно применять обобщенные алгоритмы для чтения или записи данных в объекты потоков.

Таблица 10.3. Операторы итератора `istream_iterator`

<code>istream_iterator<T> in(is);</code>	<code>in</code> читает значения типа <code>T</code> из входного потока <code>is</code>
<code>istream_iterator<T> end;</code>	Итератор после конца для итератора <code>istream_iterator</code> , читающего значения типа <code>T</code>
<code>in1 == in2</code> <code>in1 != in2</code>	<code>in1</code> и <code>in2</code> должны читать одинаковый тип. Они равны, если оба они конечные или оба связаны с тем же входным потоком
<code>*in</code>	Возвращает значение, прочитанное из потока
<code>in->mem</code>	Синоним для <code>(*in).mem</code>
<code>++in, in++</code>	Читает следующее значение из входного потока, используя оператор <code>>></code> для типа элемента. Как обычно, префиксная версия возвращает ссылку на итератор после инкремента. Постфиксная версия возвращает прежнее значение

Использование итератора `istream_iterator`

Когда создается потоковый итератор, необходимо определить тип объектов, которые итератор будет читать или записывать. Итератор `istream_iterator` использует оператор `>>` для чтения из потока. Поэтому тип, читаемый итератором `istream_iterator`, должен определять оператор ввода. При создании итератор `istream_iterator` следует связать с потоком. В качестве альтернативы итератор можно инициализировать значением по умолчанию. В результате будет создан итератор, который можно использовать как значение после конца.

```
istream_iterator<int> int_it(cin); // читает целые  
числа из cin
```

```
istream_iterator<int> int_eof; // конечное значение
```

```
итератора
ifstream in("afile");
istream_iterator<string> str_it(in); // читает
строки из "afile"
```

Для примера используем итератор `istream_iterator` для чтения со стандартного устройства ввода в вектор:

```
istream_iterator<int> in_iter(cin); // читает целые
числа из cin
```

```
istream_iterator<int> eof; // "конечный"
```

итератор `istream`

```
while (in_iter != eof) // пока есть
```

что читать

```
// постфиксный инкремент читает поток и возвращает
прежнее значение
```

```
// итератора. Обращение к значению этого итератора
предоставляет
```

```
// предыдущее значение, прочитанное из потока
```

```
vec.push_back(*in_iter++);
```

Этот цикл читает целые числа из потока `cin`, сохраняя прочитанное в вектор `vec`. На каждой итерации цикл проверяет, не совпадает ли итератор `in_iter` со значением `eof`. Этот итератор был определен как пустой итератор `istream_iterator`, который используется как конечный итератор. Связанный с потоком итератор равен конечному итератору, только если связанный с ним поток достиг конца файла или произошла ошибка ввода-вывода.

Самая трудная часть этой программы — аргумент функции `push_back()`, который использует обращение к значению и постфиксные операторы инкремента. Это выражение работает точно так же, как и другие выражения, совмещающие обращение к значению с постфиксным инкрементом (см. раздел 4.5). Постфиксный инкремент переводит поток на чтение следующего значения, но возвращает прежнее значение итератора. Это прежнее значение содержит прежнее значение, прочитанное из потока. Для того чтобы получить это значение, осуществляется обращение к значению этого итератора.

Особенно полезно то, что эту программу можно переписать так:

```
istream_iterator<int> in_iter(cin), eof; // читает
целые числа из cin
```

```
vector<int> vec(in_iter, eof); // создает
```

```
вектор vec из //  
диапазона итераторов
```

Здесь вектор `vec` создается из пары итераторов, которые обозначают диапазон элементов. Это итераторы `istream_iterator`, следовательно, диапазон получается при чтении связанного потока. Этот конструктор читает поток `cin`, пока он не встретит конец файла, или ввод, тип которого отличается от `int`. Прочитанные элементы используются для создания вектора `vec`.

Использование потоковых итераторов с алгоритмами

Поскольку алгоритмы используют функции итераторов, а потоковые итераторы поддерживают по крайней мере некоторые функции итератора, потоковые итераторы можно использовать с некоторыми из алгоритмов. Какие именно алгоритмы применимы с потоковыми итераторами, рассматривается в разделе 10.5.1. В качестве примера рассмотрим вызов функции `accumulate()` с парой итераторов `istream_iterators`:

```
istream_iterator<int> in (cin), eof;  
cout << accumulate(in, eof, 0) << endl;
```

Этот вызов создаст сумму значений, прочитанных со стандартного устройства ввода. Если ввод в этой программе будет таким:

```
23 109 45 89 6 34 12 90 34 23 56 23 8 89 23
```

то результат будет 664.

Итераторы `istream_iterator` позволяют использовать ленивое вычисление

То, что итератор `istream_iterator` связан с потоком, еще не гарантирует, что он начнет читать поток немедленно. Некоторые реализации разрешают задержать чтение потока, пока итератор не будет использован. Гарантиранно, что поток будет прочитан перед первым обращением к значению итератора. Для большинства программ не имеет никакого значения, будет ли чтение немедленным или отсроченным. Но если создается итератор `istream_iterator`, который удаляется без использования, или если необходима синхронизация чтения того же потока из двух разных объектов, то тогда придется позаботиться и о моменте чтения.

Использование итератора `ostream_iterator`

Итератор `ostream_iterator` может быть определен для любого

типа, у которого есть оператор вывода (оператор `<<`). При создании итератора `ostream_iterator` можно (необязательно) предоставить второй аргумент, определяющий символьную строку, выводимую после каждого элемента. Это должна быть символьная строка в стиле С (т.е. строковый литерал или указатель на массив с нулевым символом в конце). Итератор `ostream_iterator` следует связать с определенным потоком. Не бывает пустого итератора `ostream_iterator` или такового после конца.

Таблица 10.4. Операторы итератора `ostream_iterator`

<code>ostream_iterator<T> out(os);</code>	<code>out</code> пишет значения типа <code>T</code> в поток вывода <code>os</code>
<code>ostream_iterator<T> out(os, d);</code>	<code>out</code> пишет значения типа <code>T</code> , сопровождаемые <code>d</code> в поток вывода <code>os</code> . <code>d</code> указывает на символьный массив с нулевым символом в конце
<code>out = val</code>	Записывает <code>val</code> в поток вывода, с которым связан <code>out</code> , используя оператор <code><<</code> . Тип <code>val</code> должен быть совместим с типом, который можно писать в <code>out</code>
<code>*out, ++out, out++</code>	Эти операторы существуют, но ничего не делают с <code>out</code> . Каждый оператор возвращает итератор <code>out</code>

Итератор `ostream_iterator` можно использовать для записи последовательности значений:

```
ostream_iterator<int> out_iter(cout, " ");
for (auto e : vec)
    *out_iter++ = e; // это присвоение записывает элемент
    в cout
cout << endl;
```

Эта программа запишет каждый элемент вектора `vec` в поток `cout`, сопровождая каждый элемент пробелом. При каждом присвоении значения итератора `out_iter` происходит запись.

Следует заметить, что при присвоении итератору `out_iter` можно пропустить обращение к значению и инкремент. Таким образом, этот цикл можно переписать так:

```
for (auto e : vec)
    out_iter = e; // это присвоение записывает элемент в
    cout
cout << endl;
```

Операторы `*` и `++` ничего не делают с итератором

`ostream_iterator`, поэтому их пропуск никак не влияет на программу. Но предпочтительней писать цикл как в первом варианте. Он использует итератор единообразно с тем, как используются итераторы других типов. Этот цикл можно легко изменить так, чтобы он выполнялся итераторами других типов. Кроме того, поведение этого цикла понятней читателям нашего кода.

Чтобы не сочинять цикл самостоятельно, можно легко ввести элементы в вектор `vec` при помощи алгоритма `copy()`:

```
copy( vec.begin(), vec.end(), out_iter);
cout << endl;
```

Использование потоковых итераторов с типами класса

Итератор `istream_iterator` можно создать для любого типа, у которого есть оператор ввода (`>>`). Точно так же итератор `ostream_iterator` можно определить для любого типа, обладающего оператором вывода (`<<`). Поскольку у класса `Sales_item` есть оба оператора (ввода и вывода), итераторы ввода-вывода вполне можно использовать, чтобы переписать программу книжного магазина из раздела 1.6:

```
istream_iterator<Sales_item> item_iter(cin), eof;
ostream_iterator<Sales_item> out_iter(cout, "\n");
// сохранить первую транзакцию в sum и читать
следующую запись
Sales_item sum = *item_iter++;
while (item_iter != eof) {
    // если текущая транзакция (хранимая в item_iter)
имеет тот же ISBN
    if (item_iter->isbn() == sum.isbn())
        sum += *item_iter++; // добавить ее к sum и
читать следующую
                                // транзакцию
    else {
        out_iter = sum;      // вывести текущую сумму
        sum = *item_iter++; // читать следующую
транзакцию
    }
}
out_iter = sum; // не забыть вывести последний
набор записей
```

Эта программа использует итератор `item_iter` для чтения транзакций `Sales_item` из потока `cin`. Она использует итератор `out_iter` для записи полученной суммы в поток `cout`, сопровождая каждый вывод символом новой строки. Определив итераторы, используем итератор `item_iter` для инициализации переменной `sum` значением первой транзакции:

```
// сохранить первую транзакцию в sum и читать
следующую запись
```

```
Sales_item sum = *item_iter++;
```

Выражение осуществляет обращение к значению результата постфиксного инкремента итератора `item_iter`. Затем оно читает следующую транзакцию и инициализирует переменную `sum` значением, предварительно сохраненным в `item_iter`.

Цикл `while` выполняется до тех пор, пока поток `cin` не встретит конец файла. В цикле `while` осуществляется проверка, не относится ли содержимое переменной `sum` и только что прочитанная запись к той же книге. Если это так, то только что прочитанный объект класса `Sales_item` добавляется в переменную `sum`. Если ISBN отличаются, переменная `sum` присваивается итератору `out_iter`, который выводит текущее значение переменной `sum`, сопровождаемое символом новой строки. После вывода суммы для предыдущей книги переменной `sum` присваивается копия последней прочитанной транзакции и осуществляется инкремент итератора для чтения следующей транзакции. Цикл продолжается до конца файла или ошибки чтения. Перед завершением следует вывести значения по последней книге во вводе.

Упражнения раздела 10.4.2

Упражнение 10.29. Напишите программу, использующую потоковые итераторы для чтения текстового файла в вектор строк.

Упражнение 10.30. Используйте потоковые итераторы, а также функции `sort()` и `copy()` для чтения последовательности целых чисел со стандартного устройства ввода, их сортировки и последующего вывода на стандартное устройство вывода.

Упражнение 10.31. Измените программу из предыдущего упражнения так, чтобы она выводила только уникальные элементы. Программа должна использовать алгоритм `unique_copy()` (см. раздел 10.4.1).

Упражнение 10.32. Перепишите программу книжного магазина из раздела 1.6. Используйте вектор для хранения транзакции и различные

алгоритмы для обработки. Используйте алгоритм `sort()` с собственной функцией `compareIsbn()` из раздела 10.3.1 для упорядочивания транзакций, а затем используйте алгоритмы `find()` и `accumulate()` для вычисления суммы.

Упражнение 10.33. Напишите программу, получающую имена входного и двух выходных файлов. Входной файл должен содержать целые числа. Используя итератор `istream_iterator`, прочтайте входной файл. Используя итератор `ostream_iterator`, запишите нечетные числа в первый выходной файл. За каждым значением должен следовать пробел. Во второй файл запишите четные числа. Каждое из этих значений должно быть помещено в отдельную строку.

10.4.3. Реверсивные итераторы

Реверсивный итератор (*reverse iterator*) перебирает контейнер в обратном направлении, т.е. от последнего элемента к первому. Реверсивный итератор инвертирует смысл инкремента (и декремента): оператор `++it` переводит реверсивный итератор на предыдущий элемент, а оператор `--it` — на следующий.

Реверсивные итераторы есть у всех контейнеров, кроме `forward_list`. Для получения реверсивного итератора используют функции-члены `rbegin()`, `rend()`, `crbegin()` и `crend()`. Они возвращают реверсивные итераторы на последний элемент в контейнере и на "следующий" (т.е. предыдущий) перед началом контейнера. Подобно обычным итераторам, существуют константные и неконстантные реверсивные итераторы.

Взаимное положение этих четырех итераторов на гипотетическом векторе `vec` представлено на рис. 10.1.

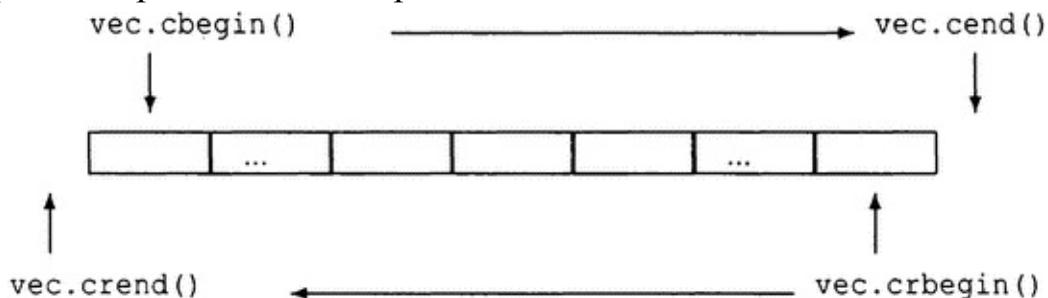


Рис. 10.1. Взаимное положение итераторов, возвращаемых функциями `begin()`/`cend()` и `rbegin()`/`crend()`

Рассмотрим, например, следующий цикл, выводящий элементы вектора

vec в обратном порядке:

```
vector<int> vec = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
// реверсивный итератор вектора (от конца к началу)  
for (auto r_iter = vec.crbegin(); // связывает  
r_iter с последним  
                                // элементом  
     r_iter != vec.crend();           // crend  
     ссылается на 1 элемент  
                                // перед 1-м  
     ++r_iter) // декремент итератора на один  
 элемент  
cout << *r_iter << endl; // выводит 9, 8, 7, ... 0
```

Хотя смысл оператора декремента реверсивного итератора может показаться неправильным, этот оператор позволяет применять для обработки контейнера стандартные алгоритмы. Например, передав функции `sort()` два реверсивных итератора, вектор можно отсортировать в порядке убывания.

```
sort(vec.begin(), vec.end()); // сортирует вектор  
vec  
                                // в "нормальном"  
порядке  
// обратная сортировка: самый маленький элемент  
располагается  
// в конце вектора vec  
sort(vec.rbegin(), vec.rend());
```

Реверсивным итераторам необходим оператор декремента

Нет ничего удивительного в том, что реверсивный итератор можно создать только из такого класса итератора, для которого определены операторы `--` и `++`. В конце концов, задача реверсивного итератора заключается в переборе последовательности назад. Кроме контейнера `forward_list`, итераторы всех стандартных контейнеров поддерживают как инкремент, так и декремент. Однако потоковые итераторы к ним не относятся, поскольку невозможно перемещать поток в обратном направлении. Следовательно, создать из потокового итератора реверсивный итератор невозможно.



Отношения между реверсивными и другими итераторами

Предположим, что существует объект `line` класса `string` (строка), содержащий разделяемый запятыми список слов. Используя функцию `find()`, можно отобразить, например, первое слово строки `line`:

```
// найти первый элемент в списке, разделенном запятыми
```

```
auto comma = find( line.cbegin(), line.cend(), ',' );
cout << string( line.cbegin(), comma ) << endl;
```

Если в строке `line` есть запятая, итератор `comma` будет указывать на нее, в противном случае он будет равен итератору, возвращаемому функцией `line.cend()`. При выводе содержимого строки от позиции `line.cbegin()` до позиции `comma` будут отображены символы от начала до запятой или вся строка, если запятых в ней нет.

Но если понадобится последнее слово в списке, то вместо обычных можно использовать реверсивные итераторы:

```
// найти последний элемент в списке, разделенном запятыми
```

```
auto rcomma = find( line.crbegin(), line.crend(), ',',
');
```

Поскольку функции `find()` в качестве аргументов передаются результаты выполнения функций `crbegin()` и `crend()`, поиск начинается с последнего символа в строке `line` в обратном порядке. По завершении поиска, если запятая найдена, итератор `rcomma` будет указывать на последнюю запятую в строке, т.е. первую запятую с конца. Если запятой нет, итератор `rcomma` будет равен итератору, возвращаемому функцией `line.crend()`.

Весьма интересна та часть, в которой осуществляется вывод найденного слова. Попытка прямого вывода создает несколько странный результат:

```
// ошибка: создаст слово в обратном порядке
cout << string( line.crbegin(), rcomma ) << endl;
```

Например, если введена строка "**FIRST, MIDDLE, LAST**", будет получен результат "TSA!"

Эта проблема проиллюстрирована на рис. 10.2. Здесь реверсивные итераторы используются для перебора строки в обратном порядке. Поэтому оператор вывода выводит строку `line` назад, начиная от `crbegin()`. Вместо этого следует выводить строку от `rcomma` и до

конца. Но итератор `rcomma` нельзя использовать непосредственно, так как это реверсивный итератор, обеспечивающий перебор от конца к началу. Поэтому необходимо преобразовать его назад в обычный итератор, перебирающий строку вперед. Для преобразования итератора `rcomma` можно применить функцию-член `base()`, которой обладает каждый реверсивный итератор.

```
// ok: получить прямой итератор и читать до конца строки
```

```
cout << string( rcomma.base( ), line.cend() ) << endl;
```

С учетом того, что введены те же данные, в результате отобразится слово "LAST", как и ожидалось.

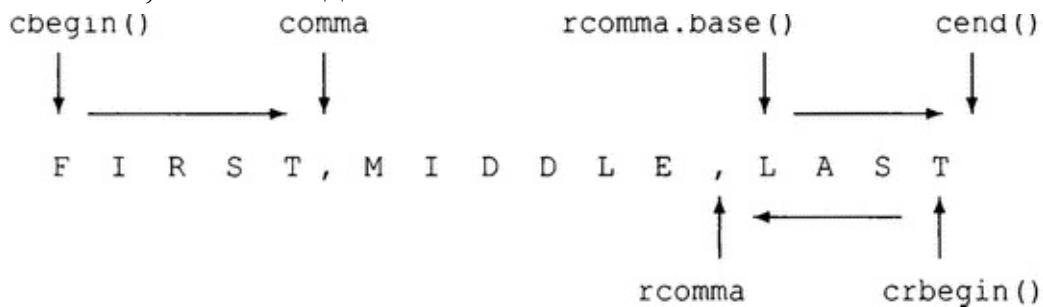


Рис. 20.2. Отношения между реверсивными и обычными итераторами

Объекты, представленные на рис. 10.2, наглядно иллюстрируют взаимоотношения между обычными и реверсивными итераторами. Например, итераторы `rcomma` и возвращаемый функцией `rcomma.base()` указывают на разные элементы, так же как и возвращаемые функциями `line.crbegin()` и `line.cend()`. Эти различия вполне обоснованы: они позволяют гарантировать возможность одинаковой обработки диапазона элементов при перемещении как вперед, так и назад.

С технической точки зрения отношения между обычными и реверсивными итераторами приспособлены к свойствам диапазона, включающего левый элемент (см. раздел 9.2.1). Дело в том, что `[line.crbegin(), rcomma)` и `[rcomma.base(), line.cend())` ссылаются на тот же элемент в строке `line`. Для этого `rcomma` и `rcomma.base()` должны возвращать соседние позиции, а не ту же позицию, как функции `crbegin()` и `cend()`.



Тот факт, что реверсивные итераторы предназначены для представления диапазонов и что эти диапазоны являются асимметричными, имеет важное последствие: при инициализации или присвоении реверсивному итератору простого итератора полученный в результате итератор не будет указывать на тот же элемент, что и исходный.

Упражнения раздела 10.4.3

Упражнение 10.34. Используйте итератор `reverse_iterator` для вывода содержимого вектора в обратном порядке.

Упражнение 10.35. Теперь отобразите элементы в обратном порядке, используя обычные итераторы.

Упражнение 10.36. Используйте функцию `find()` для поиска в списке целых чисел последнего элемента со значением 0.

Упражнение 10.37. С учетом того, что вектор содержит 10 элементов, скопируйте в список диапазон его элементов от позиции 3 до позиции 7 в обратном порядке.



10.5. Структура обобщенных алгоритмов

Фундаментальное свойство любого алгоритма — это список функциональных возможностей, которые он требует от своего итератора (итераторов). Некоторые алгоритмы, например `find()`, требуют только возможности получить доступ к элементу через итератор, прирастить итератор и сравнить два итератора на равенство. Другие, такие как `sort()`, требуют возможности читать, писать и произвольно обращаться к элементам. По своим функциональным возможностям, обязательным для алгоритмов, итераторы группируются в пять *категорий* (iterator categories), перечисленных в табл. 10.5. Каждый алгоритм определяет, итератор какого вида следует предоставить для каждого из его параметров.

Таблица 10.5. Категории итераторов

Итератор ввода	Обеспечивает чтение, но не запись; поддерживает только инкремент
Итератор вывода	Обеспечивает запись, но не чтение; поддерживает только инкремент
Прямой итератор	Обеспечивает чтение и запись; поддерживает только инкремент
Двунаправленный итератор	Обеспечивает чтение и запись; поддерживает инкремент и декремент
Итератор произвольного доступа	Обеспечивает чтение и запись; поддерживает все арифметические операции итераторов

Второй способ классификации алгоритмов (приведенный в начале этой главы) основан на том, читают ли они элементы, пишут или переупорядочивают их в последовательности. В приложении А все алгоритмы перечислены согласно этой классификации.

Алгоритмы имеют также ряд общих соглашений по передаче параметров и соглашений об именовании, рассматриваемых после категорий итераторов.



10.5.1. Пять категорий итераторов

Подобно контейнерам, для итераторов определен общий набор операций. Некоторые из них поддерживаются всеми итераторами, а другие — лишь некоторыми видами итераторов. Например, итератор `ostream_iterator` поддерживает только инкремент, обращение к значению и присвоение. Итераторы векторов, строк и двухсторонних очередей поддерживают эти операции, а также декремент, сравнение и арифметические операторы.

Таким образом, итераторы можно классифицировать на основании набора функций, которыми они обладают, а категории формируют своего рода иерархию. За исключением итераторов вывода, итераторы более высокой категории поддерживают все функции итераторов более низких категорий.

Стандарт определяет минимальную категорию для каждого параметра итератора обобщенных и числовых алгоритмов. Например, алгоритм `find()`, реализующий перебор последовательности только для чтения и в одном направлении, минимально требует только итератор ввода. Алгоритму `replace()` требуется два итератора, являющихся, по крайней мере, прямыми итераторами. Аналогично алгоритм `replace_copy()` требует прямые итераторы для своих первых двух итераторов. Его третий итератор, представляющий назначение, должен, по крайней мере, быть итератором вывода и т.д. Итератор для каждого параметра должен обладать не меньшим набором параметров, чем предусмотренный минимум. Передача итератора с меньшими возможностями недопустима.



Большинство компиляторов не заметит ошибки передачи алгоритму итератора неправильный категории.

Категории итераторов

Итератор ввода (`input iterator`) позволяет читать элементы контейнера, но записи не гарантирует. Итератор ввода обязательно должен поддерживать следующий минимум функций.

- Операторы равенства и неравенства (`==`, `!=`), используемые для сравнения двух итераторов.
- Префиксный и постфиксный инкременты (`++`), используемые для перемещения итератора.
- Оператор обращения к значению (`*`), позволяющий прочитать

элемент. Оператор обращения к значению может быть применен только к операнду, расположенному справа от оператора присвоения.

- Оператор стрелки (\rightarrow), равнозначный выражению $(*it).member$. То есть обращение к значению итератора и доступ к члену класса объекта.

Итераторы ввода могут быть использованы только последовательно. Гарантирана допустимость инкремента $*it++$, но приращение итератора ввода может сделать недопустимыми все другие итераторы в потоке. В результате нет никакой гарантии того, что можно сохранить состояние итератора ввода и исследовать элемент с его помощью. Поэтому итераторы ввода можно использовать только для однопроходных алгоритмов. Алгоритмам `find()` и `accumulate()` требуются итераторы ввода, а итератор `istream_iterator` — имеет тип итератора ввода.

Итератор вывода (`output iterator`) можно рассматривать как итератор ввода, обладающий дополнительными функциональными возможностями. Итератор вывода применяется для записи в элемент, но чтения он не гарантирует. Для итераторов вывода обязательны следующие функции.

- Префиксный и постфиксный инкременты $(++)$, используемые для перемещения итератора.
- Оператор обращения к значению $(*)$ может быть применен только к операнду, расположенному слева от оператора присвоения. Присвоение при обращении к значению итератора вывода позволяет осуществить запись в элемент.

Значение итератору вывода можно присвоить только однажды. Подобно итераторам ввода, итераторы вывода можно использовать только для однопроходных алгоритмов. Итераторы, используемые как итераторы назначения, обычно являются итераторами вывода. Например, третий параметр алгоритма `copy()` является итератором вывода. Итератор `ostream_iterator` имеет тип итератора вывода.

- *Прямой итератор* (`forward iterator`) позволяет читать и записывать данные в последовательность. Они перемещаются по последовательности только в одном направлении. Прямые итераторы поддерживают все операции итераторов ввода и вывода. Кроме того, они позволяют читать и записывать значение в тот же элемент несколько раз. Поэтому сохраненное состояние прямого итератора можно использовать. Следовательно, алгоритмы, использующие прямые итераторы, могут осуществить несколько проходов через последовательность. Алгоритму `replace()` требуется прямой итератор; итераторы контейнера `forward_list` являются прямыми итераторами.

- *Двунаправленный итератор* (bidirectional iterator) позволяет читать и записывать данные в последовательность в обоих направлениях. Кроме всех функциональных возможностей прямого итератора, двунаправленный итератор поддерживает также префиксный и постфиксный декременты (–). Алгоритму `reverse()` требуется двунаправленный итератор. Все библиотечные контейнеры, кроме `forward_list`, предоставляют итераторы, соответствующие требованиям для двунаправленного итератора.

- *Итератор прямого доступа* (random-access iterator) обеспечивает доступ к любой позиции последовательности в любой момент. Эти итераторы обладают всеми функциональными возможностями двунаправленных итераторов. Кроме того, они поддерживают операции, приведенные в табл. 3.7.

- Операторы сравнения `<`, `<=`, `>` и `>=`, позволяющие сравнить относительные позиции двух итераторов.

- Операторы сложения и вычитания (`+`, `+=`, `-` и `-=`), обеспечивающие арифметические действия между итератором и целочисленным значением. В результате получается итератор, перемещенный в контейнере вперед (или назад) на соответствующее количество элементов.

- Оператор вычитания (`-`), применяемый к двум итераторам, позволяет получить дистанцию между двумя итераторами.

- Оператор индексирования (`iter[n]`), равнозначный выражению `* (iter + n)`.

Итератор прямого доступа необходим алгоритму `sort()`. Итераторы контейнеров `array`, `deque`, `string` и `vector` являются итераторами прямого доступа, подобно указателям массива.

Упражнения раздела 10.5.1

Упражнение 10.38. Перечислите пять категорий итераторов и операции, которые каждый из них поддерживает.

Упражнение 10.39. Итератором какой категории обладает список? А вектор?

Упражнение 10.40. Итераторы какой категории нужны алгоритму `copy()`? А алгоритмам `reverse()` и `unique()`?



10.5.2. Параметрическая схема алгоритмов

Эта классификация алгоритмов основана на соглашениях об именах параметров. Понимание этих соглашений поможет в изучении новых алгоритмов: зная, что означает имя данного параметра, можно догадаться, какие операции выполняет соответствующий алгоритм. Большинство алгоритмов получают параметры в одной из четырех форм:

алг(beg, end, другие параметры) ;
алг(beg, end, dest, другие параметры) ;
алг(beg, end, beg2, другие параметры) ;
алг(beg, end, beg2, end2, другие параметры) ;

где *алг* — это имя алгоритма, а параметры *beg* и *end* обозначают исходный диапазон элементов, с которыми работает алгоритм. Хотя почти все алгоритмы получают исходный диапазон, присутствие других параметров зависит от выполняемых действий. Как правило, остальные параметры, *dest*, *beg2* и *end2*, также являются итераторами. Кроме них, некоторые алгоритмы получают дополнительные параметры, не являющиеся итераторами.

Алгоритмы с одним итератором назначения

Параметр *dest* (*destination* — назначение) — это итератор, обозначающий получателя, используемого для хранения результата. Алгоритмы подразумевают, что способны безопасно записать в последовательность назначения столько элементов, сколько необходимо.



Алгоритмы, осуществляющие запись по итератору вывода, подразумевают, что получатель достаточно велик для содержания вывода.

Если *dest* является итератором контейнера, алгоритм записывает свой результат в уже существующие элементы контейнера. Как правило, итератор *dest* связан с итератором вставки (см. раздел 10.4.1) или итератором *ostream_iterator* (см. раздел 10.4.2). Итератор вставки добавляет новые элементы в контейнер, гарантируя, таким образом, достаточную емкость. Итератор *ostream_iterator* осуществляет запись в поток вывода, а следовательно, тоже не создает никаких проблем независимо от количества записываемых элементов.

Алгоритмы с двумя итераторами, указывающими исходную последовательность

Алгоритмы, получающие один параметр (`beg2`) или два параметра (`beg2` и `end2`), используют эти итераторы для обозначения второго исходного диапазона. Как правило, для выполнения необходимых действий эти алгоритмы используют элементы второго диапазона вместе с элементами исходного.

Когда алгоритм получает параметры `beg2` и `end2`, эти итераторы обозначают весь второй диапазон. Такой алгоритм получает два полностью определенных диапазона: исходный диапазон, обозначенный итераторами `[beg, end)`, а также второй, исходный диапазон, обозначенный итераторами `[beg2, end2)`.

Алгоритмы, получающие только итератор `beg2` (но не `end2`), рассматривают итератор `beg2` как указывающий на первый элемент во втором исходном диапазоне. Конец этого диапазона не определен. В этом случае алгоритмы подразумевают, что диапазон, начинающийся с элемента, указанного итератором `beg2`, имеет, по крайней мере, такой же размер, что и диапазон, обозначенный итераторами `beg` и `end`.



Алгоритмы, получающие один параметр `beg2`, *подразумевают*, что последовательность, начинающаяся с элемента, указанного итератором `beg2`, имеет такой же размер, как и диапазон, обозначенный итераторами `beg` и `end`.



10.5.3. Соглашения об именовании алгоритмов

Кроме соглашений об именовании параметров, алгоритмы также имеют набор однозначных соглашений об именовании перегруженных версий. Эти соглашения учитывают то, как предоставляется оператор, используемый вместо принятого по умолчанию оператора `<` или `==`, а также то, используется ли алгоритм для исходной последовательности или отдельного получателя.

Некоторые алгоритмы используют перегруженные версии для передачи предиката

Как правило, перегружаются алгоритмы, которые получают предикат

для использования вместо оператора `<` или `==` и не получающие других аргументов. Одна версия функции использует для сравнения элементов оператор типа элемента, а вторая получает дополнительный параметр, являющийся предикатом, используемым вместо оператора `<` или `==`:

```
unique( beg, end); // использует для сравнения
элементов оператор ==
unique( beg, end, comp); // использует для сравнения
элементов
// предикат comp
```

Оба вызова переупорядочивают переданную последовательность, удаляя смежные повторяющиеся элементы. Первая версия для проверки на совпадение использует оператор `==` типа элемента, а вторая вызывает для этого предикат `comp`. Поскольку эти версии функции отличаются количеством аргументов, нет никакой неоднозначности (см. раздел 6.4) относительно версии вызываемой функции.

Алгоритмы с версиями `_if`

У алгоритмов, получающих значение элемента, обычно есть вторая (не перегруженная) версия, получающая предикат (см. раздел 10.3.1) вместо значения. Получающие предикат алгоритмы имеют суффикс `_if`:

```
find( beg, end, val); // найти первый экземпляр val
в исходном диапазоне
find_if( beg, end, pred); // найти первый экземпляр,
для
// которого pred
возвращает true
```

Оба алгоритма находят в исходном диапазоне первый экземпляр заданного элемента. Алгоритм `find()` ищет указанное значение, а алгоритм `find_if()` — значение, для которого предикат `pred` возвратит значение, отличное от нуля.

Эти алгоритмы предоставляют вторую именованную версию, а не перегруженную, поскольку обе версии алгоритма получают то же количество аргументов. Перегрузка была бы неоднозначна и возможна лишь в некоторых случаях. Чтобы избежать любых неоднозначностей, библиотека предоставляет отдельные именованные версии этих алгоритмов.

Различия между копирующими и не копирующими версиями

По умолчанию переупорядочивающие элементы алгоритмы

записывают результирующую последовательность в исходный диапазон. Эти алгоритмы предоставляют вторую версию, способную записывать результат по указанному назначению. Как уже упоминалось, пригодные для записи по назначению алгоритмы имеют в имени суффикс `_copy` (см. раздел 10.2.2):

```
reverse( beg, end); // обратить порядок элементов в
исходном диапазоне
reverse_copy( beg, end, dest); // скопировать
элементы по назначению в
// обратном порядке
```

Некоторые алгоритмы предоставляют и версии `_copy`, и `_if`. Эти версии получают и итератор назначения, и предикат:

```
// удаляет нечетные элементы из v1
remove_if( v1.begin(), v1.end(),
           [](int i) { return i % 2; });
// копирует только четные элементы из v1 в v2; v1
неизменен
remove_copy_if( v1.begin(), v1.end(),
                back_inserter(v2),
                [](int i) { return i % 2; });
```

Для определения нечетности элемента оба вызова используют лямбда-выражение (см. раздел 10.3.2). В первом случае нечетные элементы удаляются из самой исходной последовательности. Во втором нечетные (четные) элементы копируются из исходного диапазона в вектор `v2`.

Упражнения раздела 10.5.3

Упражнение 10.41. Исходя только из имен алгоритмов и их аргументов, опишите действия, выполняемые каждым из следующих библиотечных алгоритмов:

```
replace( beg, end, old_val, new_val);
replace_if( beg, end, pred, new_val);
replace_copy( beg, end, dest, old_val, new_val);
replace_copy_if( beg, end, dest, pred, new_val);
```

10.6. Алгоритмы, специфические для контейнеров

В отличие от других контейнеров, контейнеры `list` и `forward_list` определяют несколько алгоритмов в качестве членов. В частности, тип `list` определяют собственные версии алгоритмов `sort()`, `merge()`,

`remove()`, `reverse()` и `unique()`. Обобщенная версия алгоритма `sort()` требует итераторов произвольного доступа. В результате она не может использоваться с контейнерами `list` и `forward_list`, поскольку эти типы предоставляют двунаправленные и прямые итераторы соответственно.

Обобщенные версии других алгоритмов, определяемых типом `list`, вполне применимы со списками, но ценой производительности. Эти алгоритмы меняют элементы в исходной последовательности. Список может "поменять" свои элементы, поменяв ссылки на элементы, а не перемещая значения этих элементов. В результате специфические для списка версии этих алгоритмов могут обеспечить намного лучшую производительность, чем соответствующие обобщенные версии.

Эти специфические для списка функции приведены в табл. 10.6. В ней нет обобщенных алгоритмов, которые получают соответствующие итераторы и выполняются одинаково эффективно как для других контейнеров, так и для контейнеров `list` и `forward_list`.

Рекомендуем

Предпочтительней использовать алгоритмы-члены классов `list` и `forward_list`, а не их обобщенные версии.

Таблица 10.6. Алгоритмы-члены классов `list` и `forward_list`

Эти функции возвращают <code>void</code> .	
<code>lst.merge(lst2)</code> <code>lst.merge(lst2, comp)</code>	Объединяет элементы списков <code>lst2</code> и <code>lst</code> . Оба списка должны быть отсортированы. Элементы из списка <code>lst2</code> удаляются, и после объединения список <code>lst2</code> оказывается пустым. Возвращает тип <code>void</code> . В первой версии используется оператор <code><</code> , а во второй — указанная функция сравнения
<code>lst.remove(val)</code> <code>lst.remove_if(pred)</code>	При помощи функции <code>lst.erase()</code> удаляет каждый элемент, значение которого равно переданному значению, или для которого указанный унарный предикат возвращает значение, отличное от нуля
<code>lst.reverse()</code>	Меняет порядок элементов списка <code>lst</code> на обратный
<code>lst.sort()</code> <code>lst.sort(comp)</code>	Сортирует элементы списка <code>lst</code> , используя оператор <code><</code> или другой заданный оператор сравнения
	При помощи функции <code>lst.erase()</code> удаляет

<code>lst.unique()</code>	расположенные рядом элементы с одинаковыми
<code>lst.unique(pred)</code>	значениями. Вторая версия использует заданный бинарный предикат



Алгоритм-член `splice()`

Типы списков определяют также алгоритм `splice()`, описанный в табл. 10.7. Этот алгоритм специфичен для списочных структур данных. Следовательно, обобщенная версия этого алгоритма не нужна.

Таблица 10.7. Аргументы алгоритма-члена `splice()` классов `list` и `forward_list`

<code>lst.splice(аргументы) или flst.splice_after(аргументы)</code>	
(<code>p,</code> <code>lst2</code>)	<code>p</code> — итератор на элемент списка <code>lst</code> или итератор перед элементом списка <code>flst</code> . Перемещает все элементы из списка <code>lst2</code> в список <code>lst</code> непосредственно перед позицией <code>p</code> или непосредственно после в списке <code>flst</code> . Удаляет элементы из списка <code>lst2</code> . Список <code>lst2</code> должен иметь тот же тип, что и <code>lst</code> (или <code>flst</code>), и не может быть тем же списком
(<code>p,</code> <code>lst2,</code> <code>p2</code>)	<code>p2</code> — допустимый итератор в списке <code>lst2</code> . Перемещает элемент, обозначенный итератором <code>p2</code> , в список <code>lst</code> или элемент после обозначенного итератором <code>p2</code> в списке <code>flst</code> . Список <code>lst2</code> может быть тем же списком, что и <code>lst</code> или <code>flst</code>
(<code>p,</code> <code>lst2,</code> <code>b,</code> <code>e</code>)	<code>b</code> и <code>e</code> обозначают допустимый диапазон в списке <code>lst2</code> . Перемещает элементы в заданный диапазон из списка <code>lst2</code> . Списки <code>lst2</code> и <code>lst</code> (или <code>flst</code>) могут быть тем же списком, но итератор <code>p</code> не должен указывать на элемент в заданном диапазоне

Специфические для списка функции, изменяющие контейнер

Большинство специфических для списков алгоритмов подобны (но не идентичны) их обобщенным аналогам. Но кардинально важное различие между специфическими и обобщенными версиями в том, что специфические версии изменяют базовый контейнер. Например, специфическая версия алгоритма `remove()` удаляет указанные элементы. Специфическая версия алгоритма `unique()` удаляет второй и последующий дубликаты элемента.

Аналогично алгоритмы `merge()` и `splice()` деструктивны к своим аргументам. Например, обобщенная версия функции `merge()` запишет объединенную последовательность по заданному итератору назначения; две исходных последовательности останутся неизменны. Специфическая для списка функция `merge()` разрушит заданный список — элементы

будут удаляться из списка аргумента по мере их объединения в объект, для которого был вызван аргумент `merge()`. После объединения элементы из обоих списков продолжают существовать, но принадлежат уже одному списку.

Упражнения раздела 10.6

Упражнение 10.42. Переделайте программу, устранившую повторяющиеся слова, написанную в разделе 10.2.3, так, чтобы использовался список, а не вектор.

Резюме

Стандартная библиотека определяет порядка ста независимых от типа алгоритмов, работающих с последовательностями. Последовательности могут быть элементами контейнера библиотечного типа, встроенного массива или созданного (например, при чтении или записи в поток). Алгоритмы достигают независимости от типа, работая только с итераторами. Большинство алгоритмов получает их как первые два аргумента, обозначающие диапазон элементов. К дополнительным аргументам может относиться итератор вывода, обозначающий получателя, или другой итератор, или пара итераторов, обозначающая вторую исходную последовательность.

Итераторы подразделяются на пять категорий в зависимости от поддерживаемых ими функциональных возможностей. Категории итераторов: ввода, вывода, прямой, двунаправленный и произвольного доступа. Итератор относится к определенной категории, если он поддерживает функциональные возможности, обязательные для итератора данной категории.

Подобно категоризации итераторов по их функциональным возможностям, параметры итераторов для алгоритмов классифицируются по функциональным возможностям требуемых для них итераторов. Алгоритмам, которые только читают из последовательности, достаточно итератора ввода. Алгоритмам, которые пишут по итератору назначения, требуются итераторы вывода и т.д.

Алгоритмы никогда непосредственно не изменяют размер последовательности, с которой они работают. Они могут скопировать элементы из одной позиции в другую, но не могут самостоятельно добавить или удалить элемент.

Хотя алгоритмы не могут добавлять элементы в последовательность,

итератор вставки может создавать их. Итератор вставки связан с контейнером. При присвоении значения типа элемента контейнера итератору вставки он добавляет данный элемент в контейнер.

Контейнеры `forward_list` и `list` определяют собственные версии некоторых из обобщенных алгоритмов. В отличие от обобщенных алгоритмов, эти специфические версии изменяют переданные им списки.

Термины

Адаптер`back_inserter`. Адаптер итератора, который, получив ссылку на контейнер, создает итератор вставки, использующий функцию `push_back()` для добавления элементов в указанный контейнер.

Адаптер`front_inserter`. Адаптер итератора, который, получив ссылку на контейнер, создает итератор вставки, использующий функцию `push_front()` для добавления элементов в начало указанного контейнера.

Адаптер`inserter`. Адаптер итератора, который, получив итератор и ссылку на контейнер, создает итератор вставки, используемый функцией `insert()` для добавления элементов непосредственно перед элементом, указанным данным итератором.

Бинарный предикат (binary predicate). Предикат с двумя параметрами.

Вызываемый объект (callable object). Объект, способный быть левым операндом оператора вызова. К вызываемым объектам относятся указатели на функции, лямбда-выражения и объекты классов, определяющих перегруженные версии оператора вызова.

Двунаправленный итератор (bidirectional iterator). Поддерживает те же операции, что и прямые итераторы, плюс способность использовать оператор `--` для перемещения по последовательности назад.

Итератор`istream_iterator`. Потоковый итератор, обеспечивающий чтение из потока ввода.

Итератор`ostream_iterator`. Потоковый итератор, обеспечивающий запись в поток вывода.

Итератор ввода (input iterator). Итератор, позволяющий читать, но не записывать элементы.

Итератор вставки (insert iterator). Итератор, использующий функции контейнера для добавления элементов в данный контейнер.

Итератор вывода (output iterator). Итератор, позволяющий записывать, но не обязательно читать элементы.

Итератор перемещения (move iterator). Итератор, позволяющий

перемещать элементы, а не копировать их. Итераторы перемещения рассматриваются в главе 13.

Итератор прямого доступа (random-access iterator). Поддерживает те же операции, что и двунаправленный итератор, плюс способность использовать операторы сравнения для выяснения позиций двух итераторов относительно друг друга, а также способность осуществлять с итераторами арифметические действия, обеспечивая таким образом произвольный доступ к элементам.

Категории итераторов (iterator categories). Концептуальная организация итераторов на основании поддерживаемых ими операций. Категории итераторов составляют иерархию, в которой более мощные итераторы предоставляют те же операции, что и менее мощные. Пока итератор обеспечивает, по крайней мере, достаточный уровень операций, он вполне применим. Например, некоторым алгоритмам требуются только итераторы ввода. Такие алгоритмы могут быть применены к любому другому итератору, который обладает возможностями, не ниже, чем у итератора вывода. Алгоритмы, которым необходимы итераторы прямого доступа, применимы только для тех итераторов, которые поддерживают операции прямого доступа.

Лямбда-выражение (lambda expression). Вызываемый блок кода. Лямбы немного похожи на безымянные встраиваемые функции. Они начинается со списка захвата, позволяющего лямбда-выражению получать доступ к переменным в содержащей функции. Подобно функции, имеет список параметров (возможно пустой), тип возвращаемого значения и тело функции. У лямбда-выражения может отсутствовать тип возвращаемого значения. Если тело функции представляет собой одиничный оператор `return`, тип возвращаемого значения выводится из типа возвращаемого объекта. В противном случае типом пропущенного возвращаемого значения по умолчанию принимается `void`.

Обобщенный алгоритм (generic algorithm). Алгоритм, не зависящий от типа контейнера.

Потоковый итератор (stream iterator). Итератор, который может быть связан с потоком.

Предикат (predicate). Функция, которая возвращает значение типа `bool` (логическое) или допускающее преобразование в него. Зачастую используется обобщенными алгоритмами для проверки элементов. Используемые библиотекой предикаты являются либо унарными (получающими один аргумент), либо бинарными (получающими два аргумента).

Прямой итератор (forward iterator). Итератор, позволяющий читать и записывать элементы, но не поддерживающий оператор `--`.

Реверсивный итератор (reverse iterator). Итератор, позволяющий перемещаться по последовательности назад. У этих итераторов операторы `++` и `--` имеют противоположный смысл.

Список захвата (capture list). Часть лямбда-выражения, определяющая переменные из окружающего контекста, к которым может обращаться лямбда-выражение.

Унарный предикат (unary predicate). Предикат с одним параметром.

Функция`bind()`. Библиотечная функция, связывающая один или несколько аргументов с вызываемым выражением. Функция `bind()` определена в заголовке `functional`.

Функция`cref()`. Библиотечная функция, возвращающая копируемый объект, содержащий ссылку на константный объект типа, не допускающего копирования.

Функция`xref()`. Библиотечная функция, создающая копируемый объект из ссылки на объект типа, не допускающего копирования.

Глава 11

Ассоциативные контейнеры

Ассоциативные контейнеры кардинально отличаются от последовательных: элементы в ассоциативном контейнере хранятся и предоставляются по ключу. Элементы последовательного контейнера, напротив, хранятся и предоставляются последовательно, по их позиции в контейнере.

Хотя поведение ассоциативных контейнеров по большей части одинаково, у последовательных контейнеров оно отличается и зависит от способа использования ключей.

Ассоциативные контейнеры (*associative container*) обеспечивают быстрый поиск и предоставление элементов по ключу. Двумя первичными типами ассоциативных контейнеров являются *map* (карта) и *set* (набор). Элементами контейнера *map* являются *пары ключ-значение* (*key-value pair*): ключ выступает в роли индекса, а значение представляет собой хранимые в контейнере данные. Контейнер *set* содержит только ключи и предоставляет эффективные способы запроса на проверку наличия определенного ключа. Набор можно было бы использовать для хранения слов, которые следует проигнорировать при некой обработке текста. Карту можно использовать для словаря: слово было бы ключом, а его определение — значением.

Библиотека предоставляет восемь ассоциативных контейнеров (табл. 11.1), которые различаются по трем факторам: (1) они являются набором (*set*) или картой *map*; (2) они требуют уникальных ключей или допускает их совпадение; (3) они хранят элементы упорядочено или нет. В именах контейнеров, допускающих совпадение ключей, присутствует слово *multi*; имена контейнеров, не упорядочивающих хранимые ключи начинаются со слова *unordered*. Следовательно, *unordered_multi_set* — это набор, не требующий уникальных ключей и хранящий элементы неупорядоченными, в то время как *set* — это набор с уникальными ключами, которые хранятся упорядочено. Для организации своих элементов неупорядоченные контейнеры используют хеш-функцию. Подробно хеш-функции рассматриваются в разделе 11.4.

Таблица 11.1. Типы ассоциативных контейнеров

Элементы упорядочиваются по ключу	
<i>map</i>	Ассоциативный массив, хранящий пары ключ-значение

<code>set</code>	Контейнер, в котором ключ является значением
<code>multimap</code>	Карта, допускающая совпадение ключей
<code>multiset</code>	Набор, допускающий совпадение ключей

Неупорядоченные коллекции	
<code>unordered_map</code>	Карта, организованная по хеш-функции
<code>unordered_set</code>	Набор, организованный по хеш-функции
<code>unordered_multimap</code>	Хешированная карта; ключи могут повторяться
<code>unordered_multiset</code>	Хешированный набор; ключи могут повторяться

Типы `map` и `multimap` определены в заголовке `map.h`; классы `set` и `multiset` — в заголовке `set.h`; неупорядоченные версии контейнеров определены в заголовках `unordered_map.h` и `unordered_set.h` соответственно.



11.1. Использование ассоциативных контейнеров

Хотя большинство программистов знакомы с такими структурами данных, как векторы и списки, многие из них никогда не используют ассоциативные структуры данных. Прежде чем перейти к подробностям того, как библиотека поддерживает эти типы, имеет смысл начать с примеров использования этих контейнеров.

Карта (тип `map`) — это коллекция пар ключ-значение. Например, каждая пара может содержать имя человека как ключ и номер его телефона как значение. О такой структуре данных говорят, что она "сопоставляет имена с номерами телефонов". Тип `map` зачастую называют *ассоциативным массивом* (*associative array*). Ассоциативный массив похож на обычный массив, но его индексы не обязаны быть целыми числами. Значения в карте находят по ключу, а не по их позиции. В карте имен и номеров телефонов имя человека использовалось бы как индекс для поиска номера телефона этого человека.

В отличие от карты, набор — это просто коллекция ключей. Набор полезен тогда, когда необходимо знать, имеется ли в нем значение. Например, банк мог бы определить набор `bad_checks` для хранения имен личностей, которые выписывали фальшивые чеки. Прежде чем принять чек, этот банк запросил бы набор `bad_checks` и убедился, есть ли в нем

имя клиента.

Использование контейнера map

Классическим примером применения ассоциативного массива является программа подсчета слов:

```
// подсчитать, сколько раз каждое слово встречается
// во вводе
map<string, size_t> word_count; // пустая карта
строк и чисел
string word;
while (cin >> word)
    ++word_count[word]; // получить и прирастить
// счетчик слов
for (const auto &w : word_count) // для каждого
// элемента карты
    // отобразить результаты
    cout << w.first << " occurs " << w.second
        << ((w.second > 1) ? " times" : " time") <<
endl;
```

Эта программа читает ввод и сообщает, сколько раз встречается каждое слово.

Подобно последовательным контейнерам, ассоциативные контейнеры являются шаблонами (см. раздел 3.3). Чтобы определить карту, следует указать типы ключа и значения. В этой программе карта хранит элементы, ключи которых имеют тип `string`, а значения — тип `size_t` (см. раздел 3.5.2). При индексации карты `word_count` строка используется как индекс, а возвращаемый счетчик типа `size_t` связан с этой строкой.

Цикл `while` читает слова со стандартного устройства ввода по одному за раз. Он использует каждое слово для индексирования карты `word_count`. Если слова еще нет в карте, оператор индексирования создает новый элемент, ключом которого будет слово, а значением 0. Независимо от того, должен ли быть создан элемент, его значение увеличивается.

Как только весь ввод прочитан, серийный оператор `for` (см. раздел 3.2.3) перебирает карту выводя каждое слово и соответствующий счетчик.

При получении элемента из карты возвращается объект типа `pair` (пара), рассматриваемого в разделе 11.2.3 (стр. 545). Если не вдаваться в подробности, то `pair` — это шаблон типа, который содержит две

открытые переменные-члена по имени `first` (первый) и `second` (второй). У используемых картой пар член `first` является ключом, а `second` — соответствующим значением. Таким образом, оператор вывода должен отобразить каждое слово и связанный с ним счетчик.

Если бы эта программа была запущена для текста первого параграфа данного раздела, то вывод был бы таким:

```
Although occurs 1 time  
Before occurs 1 time  
an occurs 1 time  
and occurs 1 time  
...
```

Использование контейнера `set`

Логичным усовершенствованием создаваемой программы будет игнорирование таких распространенных слов, как "the", "and", "or" и т.д. Для хранения игнорируемых слов будет использован набор, а подсчитываться будут только те слова, которые отсутствуют в этом наборе:

```
// подсчитать, сколько раз каждое слово встречается  
во вводе  
map<string, size_t> word_count; // пустая карта  
строк и чисел  
set<string> exclude = {"The", "But", "And", "Or",  
"An", "A",  
                "the", "but", "and", "or",  
"an", "a"};  
string word;  
while (cin >> word)  
// подсчитать только не исключенные слова  
if (exclude.find(word) == exclude.end())  
    ++word_count[word]; // получить и прирастить  
счетчик слов
```

Подобно другим контейнерам, `set` является шаблоном. Чтобы определить набор, следует указать тип его элементов, которым в данном случае будет `string`. Подобно последовательным контейнерам, для элементов ассоциативного контейнера применима списочная инициализация (см. раздел 3.3.6). Набор `exclude` будет содержать 12 слов, которые следует игнорировать.

Важнейшее различие между этой программой и предыдущей в том, что перед подсчетом каждого слова оно проверяется на принадлежность к

набору исключенных. Для этого используется оператор `if`:

```
// подсчитать только не исключенные слова
if (exclude.find( word) == exclude.end() )
```

Вызов функции `find()` возвращает итератор. Если заданный ключ находится в наборе, итератор указывает на него. Если элемент не найден, функция `find()` возвращает итератор на элемент после конца. В этой версии счетчик слов изменяется, только если слово не находится в наборе `exclude`.

Если запустить эту версию для того же ввода, что и прежде, вывод будет таким:

```
Although occurs 1 time
Before occurs 1 time
are occurs 1 time
as occurs 1 time
...
...
```

Упражнения раздела 11.1

Упражнение 11.1. Опишите различия между картой и вектором.

Упражнение 11.2. Приведите пример того, когда наиболее полезен контейнер `list`, `vector`, `deque`, `map` и `set`.

Упражнение 11.3. Напишите собственную версию программы подсчета слов.

Упражнение 11.4. Усовершенствуйте свою программу так, чтобы игнорировать регистр и пунктуацию. Т.е. слова "example" и "Example", например, должны увеличить тот же счетчик.

11.2. Обзор ассоциативных контейнеров

Ассоциативные контейнеры (и упорядоченные, и неупорядоченные) поддерживают общие функции контейнеров, описанные в разделе 9.2 и перечисленные в табл. 9.2. Ассоциативные контейнеры *не поддерживают* функции, специфические для последовательных контейнеров, такие как `push_front()` или `back()`. Поскольку элементы хранятся на основании их ключа, эти операции были бы бессмысленны для ассоциативных контейнеров. Кроме того, ассоциативные контейнеры не поддерживают конструкторы и функции вставки, получающие значение элемента и его позицию.

Кроме функций, общих для всех контейнеров, ассоциативные контейнеры предоставляют некоторые функции (табл. 11.7) и псевдонимы типов (табл. 11.3), которых нет у последовательных контейнеров. Помимо этого, неупорядоченные контейнеры предоставляют функции настройки производительности их хеша, которые рассматриваются в разделе 11.4.

Итераторы ассоциативных контейнеров двунаправлены (см. раздел 10.5.1).



11.2.1. Определение ассоциативного контейнера



Как только что упоминалось, при определении карты следует указать типы ключа и значения; при определении набора задают только тип ключа, поскольку значения у него нет. Каждый из ассоциативных контейнеров имеет стандартный конструктор, который создает пустой контейнер заданного типа. Ассоциативный контейнер можно также инициализировать копией другого контейнера того же типа или диапазоном значений, тип которых может быть приведен к типу контейнера. По новому стандарту возможна также списочная инициализация элементов:

```
map<string, size_t> word_count; // пустая карта
// списочная инициализация
set<string> exclude = {"the", "but", "and", "or",
"an", "a",
```

```

        "The", "But", "And", "Or",
"An", "A"} ;
    // три элемента; authors сопоставляет фамилию с
именем
    map<string, string> authors = { {"Joyce", "James"}, 
                                    {"Austen", "Jane"}, 
                                    {"Dickens", 
"Charles"} } ;

```

Как обычно, тип инициализаторов должен быть преобразуемым в тип контейнера. Типом элемента набора является тип ключа.

При инициализации карты следует предоставить и ключ, и значение. Каждая пара ключ-значение заключается в фигурные скобки, { *ключ*, *значение* }, означая, что вместе элементы формируют единый элемент карты. Первый элемент каждой пары — это ключ, второй — значение. Таким образом, карта `authors` сопоставляет фамилии с именами и инициализируется тремя элементами.

Инициализация контейнеров `multimap` и `multiset`

Ключи в контейнерах `map` и `set` должны быть уникальными; с каждым ключом может быть сопоставлен только один элемент. У контейнеров `multimap` и `multiset` такого ограничения нет; вполне допустимо несколько элементов с тем же ключом. Например, у использованной для подсчета слов карты должен быть только один элемент, содержащий некое слово. С другой стороны, у словаря может быть несколько определений того же слова.

Следующий пример иллюстрирует различия между контейнерами с уникальными ключами и таковыми с не уникальными ключами. Сначала необходимо создать вектор целых чисел `ivec` на 20 элементов: две копии каждого из целых чисел от 0 до 9 включительно. Этот вектор будет использован для инициализации контейнеров `set` и `multiset`:

```

// определить вектор из 20 элементов, содержащий
две копии каждого
// числа от 0 до 9
vector<int> ivec;
for (vector<int>::size_type i = 0; i != 10; ++i) {
    ivec.push_back(i);
    ivec.push_back(i); // сдублировать каждое число
}

```

```
// iset содержит уникальные элементы ivec;
// miset содержит все 20 элементов
set<int> iset(ivec.cbegin(), ivec.cend());
multiset<int> miset(ivec.cbegin(), ivec.cend());
cout << ivec.size() << endl; // выводит 20
cout << iset.size() << endl; // выводит 10
cout << miset.size() << endl; // выводит 20
```

Хотя набор `iset` был инициализирован значениями всего контейнера `ivec`, он содержит только десять элементов: по одному для каждого уникального элемента вектора `ivec`. С другой стороны, контейнер `miset` содержит 20 элементов, сколько и вектор `ivec`.

Упражнения раздела 11.2.1

Упражнение 11.5. Объясните различие между картой и набором. Когда имеет смысл использовать один, а когда другой?

Упражнение 11.6. Объясните различия между набором и списком. Когда имеет смысл использовать один, а когда другой?

Упражнение 11.7. Определите карту, ключ которой является фамилией семьи, а значение — вектором имён детей. Напишите код, способный добавлять новые семьи и новых детей в существующие семьи.

Упражнение 11.8. Напишите программу, которая хранит исключенные слова в векторе, а не в наборе. Каковы преимущества использования набора?



11.2.2. Требования к типу ключа

Ассоциативные контейнеры налагают ограничения на тип ключа. Требования для ключей неупорядоченных контейнеров рассматриваются в разделе 11.4. У упорядоченных контейнеров (`map`, `multimap`, `set` и `multiset`) тип ключа должен определять способ сравнения элементов. По умолчанию для сравнения ключей библиотека использует оператор `<` типа ключа. В наборах тип ключа соответствует типу элемента; в картах тип ключа — тип первого элемента пары. Таким образом, типом ключа карты `word_count` (см. раздел 11.1) будет `string`. Аналогично типом ключа набора `exclude` также будет `string`.



Вызываемые объекты, переданные алгоритму сортировки (см. раздел 10.3.1), должны соответствовать тем же требованиям, что и ключи в ассоциативном контейнере.

Типы ключей упорядоченных контейнеров

Подобно тому, как собственный оператор сравнения можно предоставить алгоритму (см. раздел 10.3), собственный оператор можно также предоставить для использования вместо оператора `<` ключей. Заданный оператор должен обеспечить *строгое сравнение* (strict weak ordering) для типа ключа. Строгое сравнение можно считать оператором "меньше", хотя наша функция могла бы использовать более сложную процедуру. Однако самостоятельно определяемая функция сравнения должна обладать свойствами, описанными ниже.

- Два ключа не могут быть "меньше" друг друга; если ключ k_1 "меньше", чем k_2 , то k_2 никогда не должен быть "меньше", чем k_1 .
- Если ключ k_1 "меньше", чем k_2 , и ключ k_2 "меньше", чем k_3 , то ключ k_1 должен быть "меньше", чем k_3 .
- Если есть два ключа и ни один из них не "меньше" другого, то эти ключи "эквивалентны". Если ключ k_1 "эквивалентен" ключу k_2 и ключ k_2 "эквивалентен" ключу k_3 , то ключ k_1 должен быть "эквивалентен" ключу k_3 .

Если два ключа эквивалентны (т.е. если ни один не "меньше" другого), то контейнер рассматривает их как равные. С этими ключами в карте будет ассоциирован только один элемент, и любой из них предоставит доступ к тому же значению.



На практике очень важно, чтобы тип, определяющий "обычный" оператор `<`, был применим в качестве ключа.

Использование функции сравнения для типа ключа

Тип оператора, используемого контейнером для организации своих

элементов, является частью типа этого контейнера. Чтобы определить собственный оператор, следует предоставить тип этого оператора при определении типа ассоциативного контейнера. Тип оператора указывают после типа элемента в угловых скобках, используемых для указания типа определяемого контейнера.

Каждый тип в угловых скобках — это только тип. Специальный оператор сравнения (тип которого должен совпадать с типом, указанным в угловых скобках) предоставляется как аргумент конструктора при создании контейнера.

Например, невозможно непосредственно определить контейнер `multiset` объектов класса `Sales_data`, поскольку класс `Sales_data` не имеет оператора `<`. Но для этого можно использовать функцию `compareIsbn()` из упражнений раздела 10.3.1. Эта функция обеспечивает строгое сравнение на основании ISBN двух объектов класса `Sales_data`. Функция `compareIsbn()` должна выглядеть примерно так:

```
bool compareIsbn( const Sales_data &lhs, const Sales_data &rhs ) {
    return lhs.isbn() < rhs.isbn();
}
```

Чтобы использовать собственный оператор, следует определить контейнер `multiset` с двумя типами: типом ключа `Sales_data` и типом сравнения, являющимся типом указателя на функцию (см. раздел 6.7), способным указывать на функцию `compareIsbn()`. Когда определяют объекты этого типа, предоставляют указатель на функцию, которую предстоит использовать. В данном случае предоставляется указатель на функцию `compareIsbn()`:

```
// в программе может быть несколько транзакций с
тем же ISBN
// элементы bookstore упорядочены по ISBN
multiset<Sales_data, decltype(compareIsbn)*>
bookstore(compareIsbn);
```

Здесь для определения типа оператора используется спецификатор `decltype`. При использовании спецификатора `decltype` для получения указателя на функцию следует добавить символ `*` для обозначения использования указателя на заданный тип функции (см. раздел 6.7). Инициализацию `bookstore` осуществляет функция `compareIsbn()`. Это означает, что при добавлении элементов в `bookstore` они будут

упорядочены при вызове функции `compareISBN()`. Таким образом, элементы `bookstore` будут упорядочены по их члену ISBN. Аргумент конструктора можно записать как `compareISBN`, вместо `&compareISBN`, поскольку при использовании имени функции оно автоматически преобразуется в указатель, если это нужно (см. раздел 6.7). С тем же результатом можно написать `&compareISBN`.

Упражнения раздела 11.2.2

Упражнение 11.9. Определите карту, которая ассоциирует слова со списком номеров строк, в которых оно встречается.

Упражнение 11.10. Можно ли определить карту для типов `vector<int>::iterator` и `int?` А для типов `list<int>::iterator` и `int?` Если нет, то почему?

Упражнение 11.11. Переопределите `bookstore`, не используя спецификатор `decltype`.

11.2.3. Тип `pair`

Прежде чем перейти к рассмотрению действий с ассоциативными контейнерами, имеет смысл ознакомиться с библиотечным типом `pair` (пара), определенным в заголовке `utility`.

Объект типа `pair` хранит две переменные-члена. Подобно контейнерам, тип `pair` является шаблоном, позволяющим создавать конкретные типы. При создании пары следует предоставить имена двух типов, которые будут типами ее двух переменных-членов. Совпадать эти типы вовсе не обязаны.

```
pair<string, string> anon; // содержит две строки
```

```
pair<string, size_t> word_count; // содержит строку и целое число
```

```
pair<string, vector<int>> line; // содержит строку и vector<int>
```

При создании объекта пары без указания инициализирующих значений используются стандартные конструкторы типов его переменных-членов. Таким образом, пара `anon` содержит две пустые строки, а пара `line` — пустую строку и пустой вектор целых чисел. Значением переменной-члена типа `int` в паре `word_count` будет 0, а его переменная-член типа `string` окажется инициализирована пустой строкой.

Можно также предоставить инициализаторы для каждого члена пары:

```
pair<string, string> author{ "James", "Joyce" };
```

Этот код создает пару по имени `author`, инициализированную значениями "James" и "Joyce".

В отличие от других библиотечных типов, переменные-члены класса `pair` являются открытыми (см. раздел 7.2). Эти члены — `first` (первый) и `second` (второй) соответственно. К ним можно обращаться непосредственно, используя обычный точечный оператор (см. раздел 1.5.2), как, например, было сделано в операторе вывода программы подсчета слов в разделе 11.1:

```
// отобразить результаты
cout << w.first << " occurs " << w.second
    << ((w.second > 1) ? " times" : " time") <<
endl;
```

где `w` — ссылка на элемент карты. Элементами карты являются пары. В данном операторе выводится переменная-член `first` элемента, являющаяся ключом, затем переменная-член `second` элемента, являющаяся счетчиком. Библиотека определяет весьма ограниченный набор операций с парами, который приведен в табл. 11.2.

Таблица 11.2. Операции с парами

<code>pair<T1, T2> p;</code>	<code>p</code> — пара с переменными-членами типов <code>T1</code> и <code>T2</code> , инициализированными значением по умолчанию (см. раздел 3.3.1)
<code>pair<T1, T2> p(v1, v2);</code>	<code>p</code> — пара с переменными-членами типов <code>T1</code> и <code>T2</code> , инициализированными значениями <code>v1</code> и <code>v2</code> соответственно
<code>pair<T1, T2> p = { v1, v2 };</code>	Эквивалент <code>p(v1, v2)</code>
<code>make_pair(v1, v2)</code>	Возвращает пару, инициализированную значениями <code>v1</code> и <code>v2</code> . Тип пары выводится из типов значений <code>v1</code> и <code>v2</code>
<code>p. first</code>	Возвращает открытую переменную-член <code>first</code> пары <code>p</code>
<code>p. second</code>	Возвращает открытую переменную-член <code>second</code> пары <code>p</code>
<code>p1 < op < p2</code>	Операторы сравнения (<code><</code> , <code>></code> , <code><=</code> , <code>>=</code>). Сравнение осуществляется подобно упорядочиванию в словаре, т.е. оператор <code><</code> возвращает значение <code>true</code> в случае, если <code>p1. first < p2. first</code> или! <code>(p2. first < p1. first) && p1. second < p2. second</code>
<code>p1 == p2, p1 != p2</code>	Две пары равны, если их первый и второй члены соответственно равны. При сравнении используется оператор <code>==</code> хранимых элементов

Функция для создания объектов типа pair



Предположим, некая функция должна возвратить значение типа `pair`. По новому стандарту возможна списочная инициализация возвращаемого значения (см. раздел 6.3.2):

```
pair<string, int>
process( vector<string> &v) {
    // обработка v
    if (!v.empty())
        return { v.back(), v.back().size() } ; // списочная
инициализация
    else
        return pair<string, int>(); // возвращаемое
значение создано явно
}
```

Если вектор `v` не пуст, возвращается пара, состоящая из последней строки в векторе `v` и размера этой строки. В противном случае явно создается и возвращается пустая пара.

В прежних версиях языка C++ нельзя было использовать инициализаторы в скобках для возвращения типа, подобного `pair`. Вместо этого можно было написать оба оператора `return` как явно созданное возвращаемое значение:

```
if (!v.empty())
    return pair<string, int>( v.back() ,
v.back().size() );
```

В качестве альтернативы можно использовать функцию `make_pair()` для создания новой пары соответствующего типа из двух аргументов:

```
if (!v.empty())
    return make_pair( v.back() , v.back().size() );
```

Упражнения раздела 11.2.3

Упражнение 11.12. Напишите программу, читающую последовательность строк и целых чисел, сохраняя каждую прочитанную пару в объекте класса `pair`. Сохраните пары в векторе.

Упражнение 11.13. Существует по крайней мере три способа создания пар в программе предыдущего упражнения. Напишите три версии

программы, создающей пары каждым из этих способов. Укажите, какая из форм проще и почему.

Упражнение 11.14. Дополните карту фамилий семей и их детей, написанную для упражнения в разделе 11.2.1, вектором пар, содержащих имя ребенка и день его рождения.

11.3. Работа с ассоциативными контейнерами

В дополнение к типам, перечисленным в табл. 9.2 (стр. 423), ассоциативные контейнеры определяют типы, перечисленные в табл. 11.3. Они представляют типы ключа и значения контейнера.

Таблица 11.3. Псевдонимы дополнительных типов ассоциативных контейнеров

key_type	Тип ключа контейнера
mapped_type	Тип, ассоциированный с каждым ключом; только для типа map
value_type	Для наборов то же, что и key_type. Для карт — pair<const key_type, mapped type>

Для контейнеров типа set типы key_type и value_type совпадают; содержащиеся в наборе данные являются ключами. Элементами карты являются пары ключ-значение. Таким образом, каждый ее элемент — объект класса pair, содержащий ключ и связанное с ним значение. Поскольку ключ элемента изменить нельзя, ключевая часть этих пар константна:

```
set<string>::value_type v1;           // v1 - string
set<string>::key_type v2;             // v2 - string
map<string, int>::value_type v3;      // v3 -
pair<const string, int>
map<string, int>::key_type v4;        // v4 - string
map<string, int>::mapped_type v5;    // v5 - int
```

Подобно последовательным контейнерам (см. раздел 9.2.2), для доступа к члену класса, например типа map<string, int>::key_type, используется оператор области видимости.

Тип mapped_type определен только для типов карт (unordered_map, unordered_multimap, multimap и map).

11.3.1. Итераторы ассоциативных контейнеров

При обращении к значению итератора возвращается ссылка на значение типа value_type контейнера. В случае карты типом value_type является пара, переменная-член first которой содержит константный ключ, а переменная-член second — значение:

```
// получить итератор на элемент контейнера
```

```

word_count
    auto map_it = word_count.begin();
    // *map_it - ссылка на объект типа pair<const
string, size_t>
    cout << map_it->first; // отобразить ключ
элемента
    cout << " " << map_it->second; // отобразить
значение элемента
    map_it->first = "new key"; // ошибка: ключ
является константой
    ++map_it->second; // ok: значение можно изменить,
используя оператор

```



Не следует забывать, что типом `value_type` карты является `pair` и что можно изменять ее значение, но не ключ.

Итераторы наборов константы

Хотя типы наборов определяют типы `iterator` и `const_iterator`, оба типа итераторов предоставляют доступ к элементам в наборе только для чтения. Подобно тому, как нельзя изменить ключевую часть элемента карты, ключи в наборе также константы. Итератор набора можно использовать только для чтения, но не для записи значения элемента:

```

set<int> iset = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
set<int>::iterator set_it = iset.begin();
if (set_it != iset.end()) {
    *set_it = 42; // ошибка: ключи набора
    // только для чтения
    cout << *set_it << endl; // ok: позволяет читать
    // ключ
}

```

Перебор ассоциативного контейнера

Типы `map` и `set` поддерживают все функции `begin()` и `end()` из табл. 9.2. Как обычно, эти функции можно использовать для получения итераторов, позволяющих перебрать контейнер. Например, цикл вывода

результатов программы подсчета слов из раздела 11.1 можно переписать следующим образом:

```
// получить итератор на первый элемент
auto map_it = word_count.cbegin();
// сравнить текущий итератор с итератором после конца
while (map_it != word_count.cend()) {
    // обратиться к значению итератора, чтобы отобразить
    // пару ключ-значение элемента
    cout << map_it->first << " occurs "
        << map_it->second << " times" << endl;
    ++map_it; // прирастить итератор, чтобы перейти на следующий элемент
}
```

Условие цикла `while` и инкремент итератора в теле цикла такие же как в программах вывода содержимого векторов или строк. Итератор `map_it` инициализирован позицией первого элемента контейнера `word_count`. Пока итератор не равен значению, возвращенному функцией `end()`, возвращается текущий элемент, а затем происходит приращение итератора. Оператор вывода обращается к значению итератора `map_it` для получения членов пары, оставаясь в остальном тем же, что и в первоначальной программе.



Вывод этой программы имеет алфавитный порядок. При использовании итераторов для перебора контейнеров `map`, `multimap`, `set` и `multiset` они возвращают элементы в порядке возрастания ключа.

Ассоциативные контейнеры и алгоритмы

Как правило, с ассоциативными контейнерами обобщенные алгоритмы (см. главу 10) не используются. Тот факт, что ключи константны, означает невозможность передачи итераторов ассоциативных контейнеров алгоритмам, которые пишут или переупорядочивают элементы контейнеров. Таким алгоритмам нужна возможность записи в элементы. Элементы всех типов наборов константны, а у всех типов карт

константным является первый член пары.

Ассоциативные контейнеры применимы с теми алгоритмами, которые только читают элементы. Однако большинство этих алгоритмов осуществляет поиск в последовательности. Поскольку поиск элементов в ассоциативном контейнере осуществляется быстро (по ключу), как правило, не имеет смысла использовать для них обобщенный алгоритм поиска. Например, как будет продемонстрировано в разделе 11.3.5, ассоциативные контейнеры определяют функцию-член `find()`, позволяющую непосредственно выбрать элемент с заданным ключом. Для поиска элемента можно использовать обобщенный алгоритм `find()`, но он осуществляет последовательный поиск. Поэтому намного быстрее использовать функцию-член `find()` класса контейнера, чем вызывать обобщенную версию.

На практике, если это вообще происходит, ассоциативный контейнер используется с алгоритмами в качестве исходной последовательности или последовательности назначения. Например, обобщенный алгоритм `copy()` можно использовать для копирования элементов ассоциативного контейнера в другую последовательность. Точно так же адаптер `inserter` можно использовать для связи итератора вставки (см. раздел 10.4.1) с ассоциативным контейнером. Адаптер `inserter` позволяет использовать ассоциативный контейнер как место назначения для другого алгоритма.

Упражнения раздела 11.3.1

Упражнение 11.15. Каковы типы `mapped_type`, `key_type` и `value_type` карты, переменные-члены пар которой имеют типы `int` и `vector<int>`?

Упражнение 11.16. Используя итератор карты, напишите выражение, присваивающее значение элементу.

Упражнение 11.17. С учетом того, что `c` — контейнер `multiset` строк, а `v` — вектор строк, объясните следующие вызовы. Укажите, допустим ли каждый из них:

```
copy( v.begin(), v.end(), inserter( c, c.end() ) );
copy( v.begin(), v.end(), back_inserter( c ) );
copy( c.begin(), c.end(), inserter( v, v.end() ) );
copy( c.begin(), c.end(), back_inserter( v ) );
```

Упражнение 11.18. Перепишите определение типа `map_it` из цикла в данном разделы, не используя ключевое слово `auto` или `decltype`.

Упражнение 11.19. Определите переменную, инициализированную вызовом функции `begin()` контейнера `multiset` по имени `bookstore` из раздела 11.2.2. Определите тип переменной, не используя ключевое слово `auto` или `decltype`.

11.3.2. Добавление элементов

Функция-член `insert()` (табл. 11.4) добавляет один элемент или диапазон элементов в контейнер. Поскольку карта и набор (и их неупорядоченные версии) содержат уникальные ключи, попытка вставки уже присутствующего элемента не имеет никакого эффекта:

```
vector<int> ivec = { 2, 4, 6, 8, 2, 4, 6, 8};           // ivec
содержит
                                                 // восемь
элементов
set<int> set2;                                     // пустой
набор
set2.insert( ivec.cbegin(), ivec.cend() );          // set2
имеет четыре элемента
set2.insert( {1, 3, 5, 7, 1, 3, 5, 7} );           // теперь set2
имеет восемь элементов
```

Таблица 11.4. Функция `insert()` ассоциативного контейнера

<code>c.insert(v)</code> <code>c.emplace(args)</code>	<code>v</code> — объект типа <code>value_type</code> ; аргументы <code>args</code> используются при создании элемента. Элементы карты и набора вставляются (или создаются), только если элемента с данным ключом еще нет в контейнере <code>c</code> . Возвращает пару, содержащую итератор на элемент с заданным ключом и логическое значение, указывающее, был ли вставлен элемент. У контейнеров <code>multimap</code> и <code>multiset</code> осуществляется вставка (или создание) заданного элемента и возвращение итератора на новый элемент
<code>c.insert(b, e)</code> <code>c.insert(il)</code>	Итераторы <code>b</code> и <code>e</code> обозначают диапазон значений типа <code>c::value_type</code> ; <code>il</code> — заключенный в скобки список таких значений. Возвращает <code>void</code> . У карты и набора вставляются элементы с ключами, которых еще нет в контейнере <code>c</code> . У контейнеров <code>multimap</code> и <code>multiset</code> вставляются все элементы диапазона
<code>c.insert(p, v)</code> <code>c.emplace(p, args)</code>	Подобны функциям <code>insert(v)</code> и <code>emplace(args)</code> , но используют итератор <code>p</code> как подсказку для начала поиска места хранения нового элемента. Возвращает итератор на элемент с заданным ключом

Версии функции `insert()`, получающие пару итераторов или список инициализации, работают подобно соответствующим конструкторам (см. раздел 11.2.1), но добавляется только первый элемент с заданным ключом.

Добавление элементов в карту

При вставке в карту следует помнить, что типом элемента является `pair`. Зачастую объекта `pair`, подлежащего вставке, нет. В этом случае пара создается в списке аргументов функции `insert()`:

```
// четыре способа добавления слова в word_count
word_count.insert({word, 1});
word_count.insert(make_pair(word, 1));
word_count.insert(pair<string, size_t>(word, 1));
word_count.insert(map<string,
size_t>::value_type(word, 1));
```



Как уже упоминалось, по новому стандарту простейшим способом создания пары является инициализация списком аргументов в фигурных скобках. В качестве альтернативы можно вызвать функцию `make_pair()` или явно создать пару. Вот аргументы последнего вызова функции `insert()`:

```
map<string, size_t>::value_type(s, 1)
```

Он создает новый объект пары соответствующего типа для вставки в карту.

Проверка значения, возвращаемого функцией `insert()`

Значение, возвращенное функцией `insert()` (или `emplace()`), зависит от типа контейнера и параметров. Для контейнеров с уникальными ключами есть версии функций `insert()` и `emplace()`, которые добавляют один элемент и возвращают пару, сообщающую об успехе вставки. Первая переменная-член пары — итератор на элемент с заданным ключом; второй — логическое значение, указывающее на успех вставки элемента. Если такой ключ уже был в контейнере, то функция `insert()` не делает ничего, а логическая часть возвращаемого значения содержит `false`. Если такой ключ отсутствовал, то логическая часть содержит значение `true`.

Для примера перепишем программу подсчета слов с использованием функции `insert()`:

```

// более корректный способ подсчета слов во вводе
map<string, size_t> word_count; // пустая карта
строк и чисел
string word;
while (cin >> word) {
    // вставляет элемент с ключом, равным слову, и
    // значением 1;
    // если слово уже есть в word_count, insert() не
    // делает ничего
    auto ret = word_count.insert({word, 1});
    if (!ret.second) // слово уже было в word_count
        ++ret.first->second; // приращение счетчика
}

```

Для каждой строки `word` осуществляется попытка вставки со значением 1. Если слово уже находится в карте, ничего не происходит. В частности, связанный со словом счетчик остается неизменным. Если слова еще нет в карте, оно добавляется, а значение его счетчика устанавливается в 1.

Оператор `if` проверяет логическую часть возвращаемого значения. Если это значение `false`, то вставка не произошла. Следовательно, слово уже было в карте `word_count`, поэтому следует увеличить значение связанного с ним счетчика.

Еще раз о синтаксисе

Оператор приращения счетчика в этой версии программы подсчета слов трудно понять. Разобрать это выражение будет существенно проще, если сначала расставить скобки в соответствии с приоритетом (см. раздел 4.1.2) операторов:

```
+ + ( (ret. first) -> second); // эквивалентное выражение
```

Рассмотрим это выражение поэтапно.

- `ret` — пара, содержащая значение, возвращаемое функцией `insert()`.

- `ret. first` — первая переменная-член пары, на которую указывает итератор карты, с данным ключом.

- `ret. first->` — обращение к значению итератора, позволяющее получить этот элемент. Элементы карты также являются парами.

- `ret. first->second` — та часть пары элемента карты, которая является значением.

- `++ret.first->second` — инкремент этого значения.

Таким образом, оператор инкремента получает итератор для элемента с ключом слова и увеличивает счетчик, связанный с ключом, для которого не удалась попытка вставки.

Для читателей, использующих устаревший компилятор или код, предшествующий новому стандарту, объявление и инициализация пары `ret` также не совсем очевидны:

```
pair<map<string, size_t>::iterator, bool> ret =
    word_count.insert( make_pair( word, 1 ) );
```

Здесь определяется пара, вторая переменная-член которой имеет тип `bool`. Понять тип первой переменной-члена этой пары немного труднее. Это тип итератора, определенный типом `map<string, size_t>`.

Добавление элементов в контейнеры `multiset` и `multimap`

Работа программы подсчета слов зависит от того факта, что каждый ключ может присутствовать только однажды. Таким образом, с любым словом будет связан только один счетчик. Но иногда необходима возможность добавить дополнительные элементы с тем же ключом. Например, могло бы понадобиться сопоставить авторов с названиями написанных ими книг. В данном случае для каждого автора могло бы быть несколько записей, поэтому будет использован контейнер `multimap`, а не `map`. Поскольку ключи контейнеров `multi` не должны быть уникальным, функция `insert()` для них всегда вставляет элемент:

```
multimap<string, string> authors;
// добавляет первый элемент с ключом Barth, John
authors.insert( { "Barth, John", "Sot-Weed Factor" } );
// ok: добавляет второй элемент с ключом Barth, John
authors.insert( { "Barth, John", "Lost in the Funhouse" } );
```

У контейнеров, допускающих совпадение ключей, функция `insert()` получает один элемент и возвращает итератор на новый элемент. Нет никакой необходимости возвращать логическое значение, поскольку в эти контейнеры функция `insert()` всегда добавляет новый элемент.

Упражнения раздела 11.3.2

Упражнение 11.20. Перепишите программу подсчета слов из раздела 11.1 так, чтобы использовать функцию `insert()` вместо индексации.

Какая версия программы по-вашему проще? Объясните почему.

Упражнение 11.21. С учетом того, что `word_count` является картой типов `string` и `size_t`, а также того, что `word` имеет тип `string`, объясните следующий цикл:

```
while (cin >> word)
    ++word_count.insert({word, 0}).first->second;
```

Упражнение 11.22. С учетом, что `map<string, vector<int>>`, напишите типы, используемые как аргументы, и возвращаемое значение версии функции `insert()`, вставляющей один элемент.

Упражнение 11.23. Перепишите карту, хранящую вектора имен детей с ключом в виде фамилии семьи из упражнений раздела 11.2.1, так, чтобы использовался контейнер `multimap`.

11.3.3. Удаление элементов

Ассоциативные контейнеры определяют три версии функции `erase()`, описанные в табл. 11.5. Подобно последовательным контейнерам, можно удалить один элемент или диапазон элементов, передав функции `erase()` итератор или пару итераторов. Эти версии функции `erase()` подобны соответствующим функциям последовательных контейнеров: указанный элемент (элементы) удаляется и возвращается тип `void`.

Таблица 11.5. Удаление элементов ассоциативного контейнера

<code>c. erase(k)</code>	Удаляет из карты с элемент с ключом <code>k</code> . Возвращает значение типа <code>size_type</code> , указывающее количество удаленных элементов
<code>c. erase(p)</code>	Удаляет из карты с элемент, обозначенный итератором <code>p</code> . Итератор <code>p</code> должен относиться к фактически существующему элементу карты <code>c</code> , он не может быть равен итератору, возвращаемому функцией <code>c. end()</code> . Возвращает итератор на элемент после позиции <code>p</code> или <code>c. end()</code> , если итератор <code>p</code> обозначает последний элемент контейнера <code>c</code>
<code>c. erase(b, e)</code>	Удаляет элементы в диапазоне, обозначенном парой итераторов <code>b</code> и <code>e</code> . Возвращает итератор <code>e</code>

Ассоциативные контейнеры предоставляют дополнительную версию функции `erase()`, получающую аргумент типа `key_type`. Эта версия удаляет все элементы, если такие вообще имеются, с заданным ключом и возвращает количество удаленных элементов. Этую версию можно использовать для удаления определенных слов из контейнера

`word_count` прежде, чем вывести результат:

```
// удалить по ключу, возвратить количество
удаленных элементов
if (word_count.erase(removal_word))
    cout << "ok: " << removal_word << " removed\n";
else
    cout << "oops: " << removal_word << " not
found!\n";
```

Для контейнеров с уникальными ключами функция `erase()` всегда возвращает нуль или единицу. Если возвращается значение нуль, значит, удаляемого элемента не было в контейнере.

Для контейнеров с не уникальными ключами функция `erase()` возвращает количество удаленных элементов и может быть больше единицы:

```
auto cnt = authors.erase("Barth, John");
```

Если `authors` — это контейнер `multimap`, созданный в разделе 11.3.2, то переменная `cnt` будет содержать значение 2.

11.3.4. Индексация карт

Контейнеры `map` и `unordered_map` предоставляют оператор индексирования и соответствующую функцию `at()` (см. раздел 9.3.2), представленные в табл. 11.6. Типы контейнеров `set` не поддерживают индексацию, поскольку в наборе нет никакого "значения", связанного с ключом. Элементы сами являются ключами, поэтому операция "доступа к значению, связанному с ключом", бессмысленна. Нельзя индексировать контейнер `multimap` или `unordered_multimap`, поскольку с заданным ключом может быть ассоциировано несколько значений.

Таблица 11.6. Операторы индексирования контейнеров `map` и `unordered_map`

<code>c[k]</code>	Возвращает элемент с ключом <code>k</code> ; если ключа <code>k</code> нет в контейнере <code>c</code> , добавляется новый элемент, инициализированный значением с ключом <code>k</code>
<code>c. at(k)</code>	Проверяет наличие элемента с ключом <code>k</code> ; если его нет в контейнере <code>c</code> , передает исключение <code>out_of_range</code> (см. раздел 5.6)

Подобно другим использованным ранее операторам индексирования, оператор индексирования карт получает индекс (т.е. ключ) и возвращает связанное с ним значение. Однако, в отличие от других операторов

индексирования, если такого ключа еще нет, *создается новый элемент и добавляется* в карту для того ключа. Ассоциированное значение инициализируется значением по умолчанию (см. раздел 3.3.1).

Рассмотрим следующий код:

```
map <string, size_t> word_count; // пустая карта
// вставить инициализированный значением по
умолчанию элемент
// с ключом Anna; а затем установить для него
значение 1
word_count[ "Anna" ] = 1;
```

Ниже приведена имеющая место последовательность действий.

- В контейнере `word_count` происходит поиск элемента с ключом `Anna`. Элемент не найден.
- В контейнер `word_count` добавляется новая пара ключ-значение. Ключ (константная строка) содержит текст `Anna`. Значение инициализируется по умолчанию, в данном случае нулем.
- Вновь созданному элементу присваивается значение 1.

Поскольку оператор индексирования способен вставить элемент, его можно использовать только для карты, которая не является константной.



Индексация карт существенно отличается от индексации массивов или векторов: использование отсутствующего ключа приводит к *добавлению* элемента с таким ключом в карту.

Использование значения, возвращенного оператором индексирования

Иной способ индексирования карт, отличающий его от других использованных ранее операторов индексирования, влияет на тип возвращаемого значения. Обычно тип, возвращенный в результате обращения к значению итератора, и тип, возвращенный оператором индексирования, совпадают. У карт все не так: при индексировании возвращается объект типа `mapped_type`, а при обращении к значению итератора карты — объект типа `value_type` (см. раздел 11.3).

Общим у всех операторов индексирования является то, что они возвращают l-значение (см. раздел 4.1.1). Поскольку возвращается l-

значение, возможно чтение и запись в элемент:

```
cout << word_count[ "Anna"]; // получить элемент по
индексу Anna;
                                            // выводит 1
++word_count[ "Anna"];                  // получить элемент и
добавить к нему 1
cout << word_count[ "Anna"]; // получить элемент и
вывести его;
                                            // выводит 2
```



В отличие от вектора или строки, тип данных, возвращаемых оператором индексирования карты, отличается из типа, полученного при обращении к значению итератора карты.

Тот факт, что индексирование добавляет элемент, если карта его еще не содержит, позволяет создавать удивительно сжатый код, такой как цикл в программе подсчета слов (см. раздел 11.1). С другой стороны, иногда необходимо только узнать, присутствует ли элемент, но *не добавлять* его в случае отсутствия. В таких случаях не следует использовать оператор индексирования.

Упражнения раздела 11.3.4

Упражнение 11.24. Что делает следующая программа?

```
map<int, int> m;
m[ 0 ] = 1;
```

Упражнение 11.25. Сравните следующую программу с предыдущей:

```
vector<int> v;
v[ 0 ] = 1;
```

Упражнение 11.26. Какой тип применяется при индексировании карты? Какой тип возвращает оператор индексирования? Приведите конкретный пример, т.е. создайте карту, используйте типы, которые применимы для ее индексирования, а затем выявите типы, которые будет возвращать оператор индексирования.

11.3.5. Доступ к элементам

Ассоциативные контейнеры предоставляют различные способы поиска заданных элементов, описанные в табл. 11.7. Используемый способ зависит от решаемой задачи. Если нужно лишь выяснить, находится ли некий элемент в контейнере, то, вероятно, лучше использовать функцию `find()`. Для контейнеров, способных содержать только уникальные ключи, вероятно, не имеет значения, используется ли функция `find()` или `count()`. Но для контейнеров с не уникальными ключами функция `count()` выполняет больше работы: если элемент присутствует, ей все еще нужно подсчитать количество элементов с тем же ключом. Если знать количество не обязательно, лучше использовать функцию `find()`:

```
set<int> iset = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;
iset.find( 1 );      // возвращает итератор на элемент с
ключом == 1
iset.find( 11 );     // возвращает итератор ==
iset.end( )
iset.count( 1 );    // возвращает 1
iset.count( 11 );   // возвращает 0
```

Таблица 11.7. Функции поиска элементов в ассоциативном контейнере

Функции `lower_bound()` и `upper_bound()` неприменимы для неупорядоченных контейнеров. Оператор индексирования и функция `at()` применимы только для тех контейнеров `map` и `unordered_map`, которые не являются константами.

<code>c. find(k)</code>	Возвращает итератор на (первый) элемент с ключом <code>k</code> или итератор после конца, если такого элемента нет в контейнере
<code>c. count(k)</code>	Возвращает количество элементов с ключом <code>k</code> . Для контейнеров с уникальными ключами результат всегда нуль или единица
<code>c. lower_bound(k)</code>	Возвращает итератор на первый элемент, значение ключа которого не меньше, чем <code>k</code>
<code>c. upper_bound(k)</code>	Возвращает итератор на первый элемент, значение ключа которого больше, чем <code>k</code>
<code>c. equal_range(k)</code>	Возвращает пару итераторов, обозначающих элементы с ключом <code>k</code> . Если такого элемента нет, значение обеих переменных-членов равно <code>c. end()</code>

Использование функции `find()` вместо индексирования карт

Для контейнеров `map` и `unordered_map` оператор индексирования представляет простейший способ поиска значения. Но, как уже упоминалось, у оператора индексирования есть серьезный побочный эффект: если искомого ключа еще нет в карте, индексирование добавляет элемент с таким ключом. Насколько правильно такое поведение, зависит от обстоятельств. Программа подсчета слов полагалась на тот факт, что использование несуществующего ключа при индексировании приводило к вставке элемента с этим ключом и значением 0.

Иногда мы хотим знать, присутствует ли элемент с заданным ключом, не изменяя карту. Нельзя использовать оператор индексирования для определения наличия элемента, поскольку при его отсутствии оператор индексирования добавит новый элемент с таким ключом. В таких случаях следует использовать функцию `find()`:

```
if (word_count.find("foobar") == word_count.end())
    cout << "foobar is not in the map" << endl;
```

Поиск элементов в контейнерах `multimap` и `multiset`

Поиск элемента в ассоциативном контейнере с уникальными ключами довольно прост — элемент либо есть в контейнере, либо нет. Для контейнеров с не уникальными ключами все несколько сложнее, так как может существовать несколько элементов с заданным ключом. Когда в контейнере `multimap` или `multiset` содержится несколько элементов с одинаковым ключом, они располагаются в контейнере рядом.

Предположим, например, что, имея карту авторов и их книг, следует вывести все книги некоего автора. Эту задачу можно решить тремя способами. Самый очевидный из них — использовать функции `find()` и `count()`:

```
string search_item("Alain de Botton"); // искомый
автор
auto entries = authors.count(search_item); // количество записей
auto iter = authors.find(search_item); // первая
запись для этого
// автора
// перебор записей данного автора
while (entries) {
    cout << iter->second << endl; // вывод каждого
заглавия
    ++iter; // переход к следующему заглавию
```

```
--entries; // отследить количество выведенных записей  
}
```

Код начинается с вызова функции `count()`, позволяющего выяснить количество записей для данного автора, и вызова функции `find()`, позволяющего получить итератор на первый элемент с этим ключом. Количество итераций цикла `for` зависит от числа, возвращенного функцией `count()`. В частности, если функция `count()` возвратит нуль, то цикл не выполнится вообще.



Гарантируется, что перебор контейнера `multimap` или `multiset` возвратит все элементы с заданным ключом.

Другое решение на основании итератора

Задачу можно решить иначе, используя функции `lower_bound()` и `upper_bound()`. Каждая из них получает ключ и возвращает итератор. Если ключ найден в контейнере, функция `lower_bound()` возвратит итератор на первый экземпляр элемента с этим ключом, а итератор, возвращенный функцией `upper_bound()`, указывает на следующий элемент после последнего экземпляра с заданным ключом. Если таковой элемент в контейнере `multimap` отсутствует, то функции `lower_bound()` и `upper_bound()` возвратят одинаковые итераторы на позицию, в которой мог бы находиться такой ключ согласно принятому порядку. Таким образом, вызов функций `lower_bound()` и `upper_bound()` для того же ключа возвращает диапазон итераторов (см. раздел 9.2.1), обозначающий все элементы с тем же ключом.

Безусловно, возвращенный этими функциями итератор может указывать на элемент непосредственно после конца контейнера. Если искомый элемент имеет самый большой ключ в контейнере, вызов функции `upper_bound()` возвратит итератор на элемент после последнего элемента контейнера. Если элемент отсутствует и ключ является самым большим в контейнере, то вызов функции `lower_bound()` также возвратит итератор на элемент после последнего элемента контейнера.



Итератор, возвращенный функцией `lower_bound()`, может указывать, а может и не указывать на элемент с заданным ключом. Если такого элемента в контейнере нет, функция `lower_bound()` возвращает итератор на первую позицию, в которую, согласно порядку расположения элементов, мог бы быть вставлен элемент с данным ключом.

Используя эти функции, можно переписать программу следующим образом:

```
// определения authors и search_item как прежде
// итераторы beg и end обозначают диапазон
элементов данного автора
for (auto beg = authors.lower_bound(search_item),
         end = authors.upper_bound(search_item);
     beg != end; ++beg)
    cout << beg->second << endl; // вывод каждого
заглавия
```

Эта программа делает то же, что и предыдущая, использовавшая функции `count()` и `find()`, но более непосредственно. Вызов функции `lower_bound()` устанавливает итератор `beg` так, чтобы он указывал на первый элемент, соответствующий `search_item`, если он есть. Если его нет, то итератор `beg` укажет на первый элемент с ключом, большим, чем `search_item`, который может оказаться итератором после конца. Вызов функции `upper_bound()` присвоит итератору `end` позицию элемента непосредственно после последнего элемента с заданным ключом. Эти функции ничего не говорят о том, присутствует ли данный ключ в контейнере. Важный момент заключается в том, что возвращаемые значения формируют диапазон итераторов (см. раздел 9.2.1).

Если элемента с искомым ключом нет, то возвращаемые функциями `lower_bound()` и `upper_bound()` значения будут равны. Оба, по сути, укажут позицию вставки элемента с указанным ключом при сохранении текущего порядка элементов контейнера.

Если элементы с заданным ключом есть, то итератор `beg` укажет на первый такой элемент. Приращение итератора `beg` позволит перебрать элементы с этим ключом. Равенство итератора `beg` итератору `end`

свидетельствует о завершении перебора всех элементов с этим ключом.

Поскольку эти итераторы формируют диапазон, для его перебора можно использовать цикл `for`. Цикл выполняется нуль или большее количество раз, выводя записи для данного автора, если таковые вообще имеются. Если таких элементов нет, то итераторы `beg` и `end` равны и цикл не выполняется ни разу. В противном случае инкремент итератора `beg` в процессе вывода каждой связанной с данным автором записи сравняет его в конечном счете с итератором `end`.



Если функции `lower_bound()` и `upper_bound()` возвращают тот же итератор, то заданного ключа в контейнере нет.

Функция `equal_range()`

Последний способ решения этой задачи самый простой из всех: вместо функций `upper_bound()` и `lower_bound()` можно вызвать функцию `equal_range()`.

Эта функция получает ключ и возвращает пару итераторов. Если элементы с таким ключом в контейнере присутствуют, то первый итератор укажет на первый экземпляр элемента, а второй — на следующий после последнего экземпляра. Если подходящего элемента нет, то первый и второй итераторы укажут позицию, в которую этот элемент может быть вставлен.

Функцию `equal_range()` можно использовать для еще одного изменения программы:

```
// определения authors и search_item, как прежде
// pos содержит итераторы, обозначающие диапазон
// элементов
// с заданным ключом
for (auto pos = authors.equal_range(search_item);
     pos.first != pos.second; ++pos.first)
    cout << pos.first->second << endl; // вывод
каждого заглавия
```

Эта программа очень похожа на предыдущую, где использовались функции `upper_bound()` и `lower_bound()`. Для хранения диапазона итераторов вместо локальных переменных `beg` и `end` используется пара,

возвращенная функцией `equal_range()`. Переменная-член `first` этой пары содержит тот же итератор, который возвратила бы функция `lower_bound()`, а переменная-член `second` — итератор, который возвратила бы функция `upper_bound()`. Таким образом, в этой программе значение `pos.first` эквивалентно значению `beg`, а `pos.second` — значению `end`.

Упражнения раздела 11.3.5

Упражнение 11.27. Для решения каких видов задач используется функция `count()`? Когда вместо нее можно использовать функцию `find()`?

Упражнение 11.28. Определите и инициализируйте переменную, содержащую результат вызова функции `find()` для карты строк и векторов целых чисел.

Упражнение 11.29. Что возвращают функции `upper_bound()`, `lower_bound()` и `equal_range()`, когда им передается ключ, отсутствующий в контейнере?

Упражнение 11.30. Объясните значение операнда `pos.first->second`, использованного в выражении вывода последней программы данного раздела.

Упражнение 11.31. Напишите программу, определяющую контейнер `multimap` авторов и их работ. Используйте функцию `find()` для поиска элемента и его удаления. Убедитесь в корректности работы программы, когда искомого элемента нет в карте.

Упражнение 11.32. Используя контейнер `multimap` из предыдущего упражнения, напишите программу вывода списка авторов и их работ в алфавитном порядке.

11.3.6. Карта преобразования слов

Завершим этот раздел программой, иллюстрирующей создание, поиск и перебор карты. Программа получает одну строку и преобразует ее в другую. Программе передаются два файла: первый содержит правила, используемые для преобразования текста во втором файле. Каждое правило состоит из слова, которое может встретиться во входном файле, и фразы, используемой вместо него. Идея в том, чтобы, встретив во вводе некое слово, заменить его соответствующей фразой. Второй файл содержит преобразуемый текст.

Вот содержимое файла преобразования слов.

```
brb be right back  
k okay?  
y why  
r are  
u you  
pic picture  
thk thanks!  
18r later
```

Подлежащий преобразованию текст таков:

```
where r u  
y dont u send me a pic  
k thk 18r
```

Программа должна создать следующий вывод:

```
where are you  
why dont you send me a picture  
okay? thanks! later
```

Программа преобразования слова

Решение подразумевает использование трех функций. Функция `word_transform()` будет осуществлять общую обработку. Потребуются два аргумента типа `ifstream`: первый будет связан с файлом преобразования слов, а второй — с текстовым файлом, который предстоит преобразовать. Функция `buildMap()` будет читать файл правил преобразования и создавать элемент карты для каждого слова и результата его преобразования. Функция `transform()` получит строку и, если она есть в карте, возвратит результат преобразования.

Давайте начнем с определения функции `word_transform()`. Важнейшие ее части — вызовы функций `buildMap()` и `transform()`:

```
void word_transform(ifstream &map_file, ifstream &input) {  
    auto trans_map = buildMap(map_file); // хранит преобразования  
    string text; // содержит каждую строку из ввода  
    while (getline(input, text)) { // читать строку из ввода  
        istringstream stream(text); // читать каждое слово  
        string word;  
        bool firstword = true; // контролирует вывод
```

пробела

```
    while (stream >> word) {  
        if (firstword)  
            firstword = false;  
        else  
            cout << " "; // вывод пробела между словами  
            // transform( ) возвращает свой первый аргумент  
или  
        // результат преобразования  
        cout << transform(word, trans_map); // вывод  
результата  
    }  
    cout << endl; // обработка текущей строки ввода  
окончена  
}  
}
```

Функция начинается вызовом функции `buildMap()`, создающим карту преобразования слов. Результат сохраняется в карте `trans_map`. Остальная часть функции обрабатывает входной файл. Цикл `while` использует функцию `getline()` для чтения входного файла по одной строке за раз. Построчно чтение осуществляется для того, чтобы строки вывода заканчивались там же, где и строки входного файла. Для получения слов каждой строки используется вложенный цикл `while`, использующий строковый поток `istringstream` (см. раздел 8.3) для обработки каждого слова текущей строки.

Внутренний цикл `while` выводит результат, используя логическую переменную `firstword`, чтобы решить, выводить ли пробел. Вызов функции `transform()` получает подлежащее выводу слово. Значение, возвращенное функцией `transform()`, будет либо исходным словом строки, либо соответствующим ему преобразованием из карты `transmap`.

Создание карты преобразования

Функция `buildMap()` читает переданный ей файл и создает карту преобразований.

```
map<string, string> buildMap(ifstream &map_file) {  
    map<string, string> trans_map; // хранит  
преобразования  
    string key; // слово для преобразования
```

```

    string value; // фраза, используемая вместо него
    // прочитать первое слово в ключ, а остальную
    часть строки в значение
    while (map_file >> key && getline(map_file,
value))
        if (value.size() > 1) // проверить, есть ли
преобразование
            trans_map[key] = value.substr(1); // убрать
предваряющий
                                            // пробел
        else
            throw runtime_error("no rule for " + key);
        return trans_map;
    }

```

Каждая строка файла `map_file` соответствует правилу. Каждое правило — это слово, сопровождаемое фразой, способной содержать несколько слов. Для чтения слов, преобразуемых в ключи, используется оператор `>>` и функция `getline()` для чтения剩余部分 of the string in value. Поскольку функция `getline()` не отбрасывает предваряющие пробелы (см. раздел 3.2.2), необходимо убрать пробел между словом и соответствующим ему правилом. Прежде чем сохранить преобразование, осуществляется проверка наличия в нем хотя бы одного символа. Если это так, то происходит вызов функции `substr()` (см. раздел 9.5.1), позволяющий устраниить пробел, отделяющий фразу преобразования от соответствующего ему слова, и сохранить эту подстроку в карте `trans_map`.

Обратите внимание на использование оператора индексирования при добавлении пары ключ-значение. При этом неявно игнорируется происходящее при повторении слова в файле преобразования. Если слово повторяется несколько раз, то в карте `trans_map` окажется последняя соответствующая фраза. По завершении цикла `while` карта `trans_map` содержит все данные, необходимые для преобразования ввода.

Осуществление преобразования

Фактическое преобразование осуществляется функция `transform()`. Ее параметры — ссылки на преобразуемую строку и карту преобразования. Если переданная строка находится в карте, функция `transform()` возвращает соответствующую ей фразу преобразования. Если переданной

строки в карте нет, функция `transform()` возвращает свой аргумент:

```
const string &
transform( const string &s, const map<string,
string> &m) {
    // фактическая работа карты; это основная часть
программы
    auto map_it = m.find(s);
    // если слово есть в карте преобразования
    if (map_it != m.cend())
        return map_it->second; // использовать замену
слова
    else
        return s; // в противном случае возвратить
исходное слово
}
```

Код начинается с вызова функции `find()`, позволяющего определить, находится ли данная строка в карте. Если это так, то функция `find()` возвращает итератор на соответствующий элемент. В противном случае функция `find()` возвращает итератор на элемент после конца. Если элемент найден, обращение к значению итератора возвращает пару, содержащую ключ и значение этого элемента (см. раздел 11.3). Функция возвращает значение переменной-члена `second` этой пары, являющееся преобразованной фразой, используемой вместо строки `s`.

Упражнения раздела 11.3.6

Упражнение 11.33. Реализуйте собственную версию программы преобразования слов.

Упражнение 11.34. Что будет, если в функции `transform()` вместо функции `find()` использовать оператор индексирования?

Упражнение 11.35. Что будет (если будет) при таком изменении функции `buildMap()`:

```
trans_map[key] = value.substr(1);
as trans_map.insert({key, value.substr(1)})?
```

Упражнение 11.36. Текущая версия программы не проверяет допустимость входного файла. В частности, она подразумевает, что все правила в файле преобразований корректны. Что будет, если строка в этом файле содержит ключ, один пробел и больше ничего? Проверьте свой ответ на текущей версии программы.



11.4. Неупорядоченные контейнеры



Новый стандарт определяет четыре *неупорядоченных ассоциативных контейнера* (unordered container). Вместо оператора сравнения для организации своих элементов эти контейнеры используют *хеш-функцию* (hash function) и оператор == типа ключа. Неупорядоченный контейнер особенно полезен, когда имеющийся тип ключа не дает очевидных отношений для упорядочивания элементов. Эти контейнеры полезны также в приложениях, где цена упорядочивания элементов высока.

Хотя в принципе хеширование обеспечивает лучшую среднюю производительность, достижение хороших результатов на практике зачастую требует серьезной проверки производительности и настройки. В результате обычно проще (а зачастую и производительней) использовать упорядоченный контейнер.



Используйте неупорядоченный контейнер, если тип ключа принципиально неупорядочен или если проверка производительности свидетельствует о проблеме, решить которую позволит только хеширование.

Использование неупорядоченного контейнера

Кроме функций управления хешированием, неупорядоченные контейнеры предоставляют те же функции (`find()`, `insert()` и т.д.), что и упорядоченные контейнеры. Это значит, что функции, использовавшиеся для контейнеров `map` и `set`, применимы также к контейнерам `unordered_map` и `unordered_set`. Аналогично неупорядоченные контейнеры имеют версии с не уникальными ключами.

В результате вместо соответствующего упорядоченного контейнера можно обычно использовать неупорядоченный контейнер, и наоборот. Но поскольку элементы хранятся неупорядочено, вывод программы, использующей неупорядоченный контейнер, будет (обычно) отличаться от такого при использовании упорядоченного контейнера.

Например, первоначальную программу подсчета слов из раздела 11.1 можно переписать так, чтобы использовать контейнер `unordered_map`:

```
// подсчет слов, но слова не в алфавитном порядке
unordered_map<string, size_t> word_count;
string word;
while (cin >> word)
    ++word_count[word]; // получить и прирастить
// счетчик слов
for (const auto &w : word_count) // для каждого
// элемента карты
    // отобразить результаты
    cout << w.first << " occurs " << w.second
        << ((w.second > 1) ? " times" : " time") <<
endl;
```

Единственное различие между этой программой и первоначальной заключается в типе `word_count`. Если запустить эту версию для того же ввода, то получится то же количество для каждого слова.

```
containers occurs 1 time
use occurs 1 time
can occurs 1 time
examples occurs 1 time
...
```

Но вывод вряд ли будет в алфавитном порядке.

Управление ячейками

Неупорядоченные контейнеры организованы как коллекция ячеек, каждая из которых содержит любое количество элементов. Для группировки элементов в ячейки контейнер использует хеш-функцию. Для доступа к элементу контейнер сначала вычисляет хеш-код элемента, указывающий, где искать ячейку. Контейнер помещает все элементы с одинаковым значением хеш-функции в ту же ячейку. Если контейнер допускает несколько элементов с одинаковым ключом, то все элементы с этим ключом будут находиться в той же ячейке. В результате производительность неупорядоченного контейнера зависит от качества его хеш-функции, количества и размера его ячеек.

Хеш-функция всегда возвращает тот же результат, будучи вызванной с тем же аргументом. В идеале хеш-функция сопоставляет также каждое конкретное значение с уникальной ячейкой. Однако хеш-функция может сопоставить элементы с разными ключами с той же ячейкой. Когда ячейка

содержит несколько элементов, поиск искомого среди них осуществляется последовательно. Как правило, вычисление хеш-кода элемента и поиск его ячейки осуществляется очень быстро. Но если в ячейке много элементов, то для поиска определенного элемента может понадобиться много сравнений.

Неупорядоченные контейнеры предоставляют набор перечисленных в табл. 11.8 функций, позволяющих управлять ячейками. Эти функции-члены позволяют запрашивать состояние контейнера и реорганизовать его по мере необходимости.

Таблица 11.8. Функции управления неупорядоченным контейнером

Взаимодействие с ячейками	
c. bucket_count()	Количество используемых ячеек
c. max_bucket_count()	Наибольшее количество ячеек, которое может содержать данный контейнер
c. bucket_size(n)	Количество элементов в ячейке n
c. bucket(k)	Ячейка, в которой следует искать элементы с ключом k
Перебор ячеек	
local_iterator	Тип итератора, способный обращаться к элементам в ячейке
const_local_iterator	Константная версия итератора ячейки
c. begin(n), c. end(n)	Итераторы на первый и следующий после последнего элементы ячейки n
c. cbegin(n), c. cend(n)	Возвращают итератор const_local_iterator
Политика хеша	
c. load_factor()	Среднее количество элементов на ячейку. Возвращает тип float
c. max_load_factor()	Средний размер ячейки, который пытается поддерживать контейнер c. Контейнер c добавляет ячейки, чтобы сохранить соотношение load_factor <= max_load_factor. Возвращает тип float
c. rehash(n)	Реорганизует хранилище так, чтобы bucket_count >= n и bucket_count > size/max_load_factor
c. reserve(n)	Реорганизует контейнер c так, чтобы он мог содержать n элементов без вызова функции rehash()

Требования к типу ключа неупорядоченных контейнеров

По умолчанию для сравнения элементов неупорядоченные контейнеры

используют оператор `==` типа ключа. Они также используют объект типа `hash<key_type>` при создании хеш-кода для каждого элемента. Библиотека поставляет также версии шаблона *хеша* для встроенных типов, включая указатели. Она определяет также шаблон `hash` для некоторых из библиотечных типов, включая строки и интеллектуальные указатели, которые рассматривались в главе 12. Таким образом, можно непосредственно создать неупорядоченный контейнер, ключ которого имеет один из встроенных типов (включающий типы указателей) либо тип `string` или интеллектуального указателя.

Однако нельзя непосредственно определить неупорядоченный контейнер, использующий для ключа собственные типы классов. В отличие от контейнеров, шаблон хеша нельзя использовать непосредственно. Вместо этого придется предоставить собственную версию шаблона `hash`. Это будет описано в разделе 16.5.

Вместо хеша по умолчанию можно применить стратегию, подобную используемой при переопределении заданного по умолчанию оператора сравнения ключей упорядоченных контейнеров (см. раздел 11.2.2). Чтобы использовать тип `Sales_data` для ключа, необходимо предоставить функцию для замены оператора `==` и вычисления хеш-кода. Начнем с определения этих функций:

```
size_t hasher( const Sales_data &sd) {
    return hash<string>()( sd.isbn() );
}
bool eqOp( const Sales_data &lhs, const Sales_data
&rhs) {
    return lhs.isbn() == rhs.isbn();
}
```

Чтобы создать хеш-код для переменной-члена ISBN, функция `hasher()` использует объект библиотечного типа `hash` для типа `string`. Точно так же функция `eqOp()` сравнивает два объекта класса `Sales_data`, сравнивая их ISBN.

Эти функции можно также использовать для определения контейнера `unordered_multiset` следующим образом:

```
using SD_multiset = unordered_multiset<Sales_data,
                                         decltype( hasher)*,
                                         decltype( eqOp)*>;
// аргументы - размер ячееки, указатель на оператор
равенства и
```

```
// хеш-функцию
SD_multiset bookstore( 42, hasher, eqOp);
```

Чтобы упростить объявление `bookstore`, определим сначала псевдоним типа (см. раздел 2.5.1) для контейнера `unordered_multiset`, у хеша и оператора равенства которого есть те же типы, что и у функций `hasher()` и `eqOp()`. Используя этот тип, определим `bookstore`, передав указатели на функции, которые он должен использовать.

Если у класса есть собственный оператор `==`, можно переопределить только хеш-функцию:

```
// использовать FooHash для создания хеш-кода;
// у Foo должен быть оператор ==
unordered_set<Foo, decltype(FooHash)*> fooSet( 10,
FooHash);
```

Упражнения раздела 11.4

Упражнение 11.37. Каковы преимущества неупорядоченного контейнера по сравнению с упорядоченной версией этого контейнера? Каковы преимущества упорядоченной версии?

Упражнение 11.38. Перепишите программы подсчета слов (см. раздел 11.1) и преобразования слов (см. раздел 11.3.6) так, чтобы использовать контейнер `unordered_map`.

Резюме

Ассоциативные контейнеры обеспечивают эффективный поиск и возвращение элементов по ключу. Использование ключа отличает ассоциативные контейнеры от последовательных, в которых к элементам обращаются по позиции.

Существует восемь ассоциативных контейнеров со следующими свойствами.

- Кarta хранит пары ключ-значение; набор хранит только ключи.
- Есть контейнеры с уникальными ключами и с не уникальными.
- Ключи могут храниться упорядоченными или нет.

Упорядоченные контейнеры используют функцию сравнения для упорядочивания элементов по ключу. По умолчанию для сравнения используется оператор `<` типа ключа. Неупорядоченные контейнеры используют для организации своих элементов оператор `==` типа ключа и объект типа `hash<key_type>`.

Имена контейнеров с не уникальными ключами включают слово

`multi`; а имена контейнеров, использующих хеширование, начинаются словом `unordered`. Контейнер `set` — это упорядоченная коллекция, каждый ключ которой уникален; контейнер `unordered_multiset` — это неупорядоченная коллекция, ключи которой могут повторяться.

Ассоциативные контейнеры имеют много общих операций с последовательными контейнерами. Но ассоциативные контейнеры определяют некоторые новые функции и переопределяют значение и типы возвращаемого значения некоторых функций, общих для последовательных и ассоциативных контейнеров. Различия в функциях отражают способ использования ключей в ассоциативных контейнерах.

Итераторы упорядоченных контейнеров обеспечивают доступ к элементам по ключу. Элементы с тем же ключом хранятся рядом друг с другом и в упорядоченных, и в неупорядоченных контейнерах.

Термины

Ассоциативный контейнер (associative container). Тип, содержащий коллекцию объектов и обеспечивающий эффективный поиск по ключу.

Ассоциативный массив (associative array). Массив, элементы которого проиндексированы по ключу, а не по позиции. Таким образом, массив сопоставляет (ассоциирует) ключ со значением.

Контейнер`map` (карта). Ассоциативный контейнер, аналогичный ассоциативному массиву. Подобно типу `vector`, тип `map` является шаблоном класса. Но при создании карты необходимо указать два типа: тип ключа и тип связанного с ним значения. В контейнере `map` ключи уникальны, они не повторяются. Каждый ключ связан с определенным значением. Обращение к значению итератора карты возвращает объект типа `pair`, который содержит константный ключ и связанное (ассоциированное) с ним значение.

Контейнер`multimap`. Ассоциативный контейнер, подобный контейнеру `map`, но способный содержать одинаковые ключи.

Контейнер`multiset`. Ассоциативный контейнер, который содержит только ключи. В отличие от набора, способен содержать одинаковые ключи.

Контейнер`set` (набор). Ассоциативный контейнер, который содержит только ключи. Ключи в контейнере `set` не могут совпадать.

Контейнер`unordered_map`. Контейнер, элементы которого являются парами ключ-значение. Допустим только один элемент на ключ.

Контейнер`unordered_multimap`. Контейнер, элементы которого являются парами ключ-значение. Допустимо несколько элементов на ключ.

Контейнер`unordered_multiset`. Контейнер, хранящий ключи. Допустимо несколько элементов на ключ.

Контейнер`unordered_set`. Контейнер, хранящий ключи. Допустим только один элемент на ключ.

Неупорядоченный контейнер (`unordered container`). Ассоциативные контейнеры, использующие хеширование, а не сравнение ключей для хранения и доступа к элементам. Эффективность этих контейнеров зависит от качества хеш-функции.

Оператор `*`. Оператор обращения к значению, примененный к итератору контейнера `map`, `set`, `multimap` или `multiset`, возвращает объект типа `value_type`. Обратите внимание на то, что типом `value_type` контейнера `map` и `multimap` является пара (pair).

Оператор `[]`. Оператор индексирования, примененный к контейнеру `map`, получает индекс, типом которого должен быть `key_type` (или тип, допускающий преобразование в него). Возвращает значение типа `mapped_type`.

Строгое сравнение (`strict weak ordering`). Отношения между ключами ассоциативного контейнера. При строгом сравнении можно сравнить два любых значения и выяснить, которое из них меньше. Если ни одно из значений не меньше другого, они считаются равными.

Тип`key_type`. Тип, определенный в шаблоне ассоциативного контейнера, которому соответствует тип ключей, используемых для сохранения и возвращения значения. У контейнера `map` тип `key_type` используется для индексации. У контейнера `set` типы `key_type` и `value_type` совпадают.

Тип`mapped_type`. Тип, определенный в шаблонах ассоциативных контейнеров `map` и `multimap`, которому соответствует тип хранимых значений.

Тип`pair` (пара). Тип, объект которого содержит две открытые переменные-члена по имени `first` (первый) и `second` (второй). Тип `pair` является шаблоном, при создании класса которого указывают два типа: тип первого и тип второго элемента.

Тип`value_type`. Тип элемента, хранимого в контейнере. У контейнеров `set` и `multiset` типы `value_type` и `key_type` совпадают. У контейнеров `map` и `multimap` этот тип представляет собой пару, первый элемент которой (`first`) имеет тип `const key_type`, а второй

(second) — тип mapped_type.

Хеш (hash). Специальный библиотечный шаблон, который используют неупорядоченные контейнеры для управления позицией элементов.

Хеш-функция (hash function). Функция, сопоставляющая значения заданного типа с целочисленными значениями (size_t). Равные значения должны сопоставляться с равными целыми числами; неравные значения должны сопоставляться с неравными целыми числами, если это возможно.

Глава 12

Динамическая память

Написанные до сих пор программы использовали объекты, имевшие четко определенную продолжительность существования. Глобальные объекты создаются при запуске программы и освобождаются по завершении выполнения программы. Локальные автоматические объекты создаются при входе в блок, где они определены, и удаляются при выходе из него. Статические локальные объекты создаются перед их первым использованием и удаляются по завершении программы.

В дополнение к автоматическим и статическим объектам язык C++ позволяет создавать объекты динамически. Продолжительность существования объектов, созданных динамически, не зависит от того, где они созданы; они существуют, пока не будут освобождены явно.

Процесс освобождения динамических объектов оказывается удивительно богатым источником ошибок. Чтобы сделать использование динамических объектов безопасней, библиотека определяет два типа интеллектуальных указателей, управляющих динамическим созданием объектов. Интеллектуальные указатели гарантируют, что объекты, на которые они указывают, будут автоматически освобождены в соответствующий момент.

До сих пор наши программы использовали только статические объекты или объекты, располагаемые в стеке. Статическая память используется для локальных статических переменных (см. раздел 6.1.1), для статических переменных-членов классов (см. раздел 7.6), а также для переменных, определенных вне функций. Стек используется для нестатических объектов, определенных в функциях. Объекты, расположенные в статической памяти или в стеке, автоматически создаются и удаляются компилятором. Объекты из стека существуют, только пока выполняется блок, в котором они определены; статические объекты создаются прежде, чем они будут использованы, и удаляются по завершении программы.

Кроме статической памяти и стека, у каждой программы есть также пул памяти, которую она может использовать. Это *динамическая память* (free store) или *распределяемая память* (heap). Программы используют распределяемую память для объектов, называемых *динамически созданными объектами* (dynamically allocated object), место для которых программа резервирует во время выполнения. Программа сама

контролирует продолжительность существования динамических объектов; наш код должен явно освобождать такие объекты, когда они больше не нужны.



Хотя динамическая память иногда необходима, ее корректное освобождение зачастую довольно сложно.

12.1. Динамическая память и интеллектуальные указатели

Для управления динамической памятью в языке C++ используются два оператора: *оператор new*, который резервирует (а при необходимости и инициализирует) объект в динамической памяти и возвращает указатель на него; и *оператор delete*, который получает указатель на динамический объект и удаляет его, освобождая зарезервированную память.

Работа с динамической памятью проблематична, поскольку на удивление сложно гарантировать освобождение памяти в нужный момент. Если забыть освобождать память, то появится утечка памяти, если освободить область памяти слишком рано, пока еще есть указатели на нее, то получится указатель на не существующую более область памяти.



Чтобы сделать использование динамической памяти проще (и безопасный), новая библиотека предоставляет два типа *интеллектуальных указателей* (smart pointer) для управления динамическими объектами.

Интеллектуальный указатель действует, как обычный указатель, но с важным дополнением: автоматически удаляет объект, на который он указывает. Новая библиотека определяет два вида интеллектуальных указателей, отличающихся способом управления своими базовыми указателями: указатель `shared_ptr` позволяет нескольким указателям указывать на тот же объект, а указатель `unique_ptr` — нет. Библиотека определяет также сопутствующий класс `weak_ptr`, являющийся второстепенной ссылкой на объект, управляемый указателем `shared_ptr`. Все три класса определены в заголовке `memory`.

12.1.1. Класс `shared_ptr`

C++
11

Подобно векторам, интеллектуальные указатели являются шаблонами (см. раздел 3.3). Поэтому при создании интеллектуального указателя следует предоставить дополнительную информацию — в данном случае тип, на который способен указывать указатель. Подобно векторам, этот тип указывают в угловых скобках, следующих за именем типа определяемого интеллектуального указателя:

```
shared_ptr<string> p1;           // shared_ptr может
указывать на строку
shared_ptr<list<int>> p2;       // shared_ptr может
указывать на
                                // список целых чисел
```

Инициализированный по умолчанию интеллектуальный указатель хранит нулевой указатель (см. раздел 2.3.2). Дополнительные способы инициализации интеллектуального указателя рассматриваются в разделе 12.1.3.

Интеллектуальный указатель используется теми же способами, что и обычный указатель. Обращение к значению интеллектуального указателя возвращает объект, на который он указывает. Когда интеллектуальный указатель используется в условии, результат проверки может засвидетельствовать, не является ли он нулевым:

```
// если указатель p1 не нулевой и не указывает на
пустую строку
if (p1 && p1->empty())
    *p1 = "hi"; // обратиться к значению p1, чтобы
присвоить ему
                                // новое значение строки
```

Список общих функций указателей `shared_ptr` и `unique_ptr` приведен в табл. 12.1. Функции, специфические для указателя `shared_ptr`, перечислены в табл. 12.2.

Таблица 12.1. Функции, общие для указателей `shared_ptr` и `unique_ptr`

<code>shared_ptr<T> sp</code>	Нулевой интеллектуальный указатель, способный указывать на
-------------------------------------	--

<code>unique_ptr<T> up</code>	объекты типа T
<code>p</code>	При использовании указателя p в условии возвращается значение <code>true</code> , если он указывает на объект
<code>*p</code>	Обращение к значению указателя p возвращает объект, на который он указывает
<code>p->mem</code>	Синоним для (<code>*p</code>). <code>mem</code>
<code>p.get()</code>	Возвращает указатель, хранимый указателем p. Используйте его осторожно, поскольку объект, на который он указывает, может прекратить существование после удаления его интеллектуальным указателем
<code>swap(p, q)</code> <code>p.swap(q)</code>	Обменивает указатели в p и q

Таблица 12.2. Функции, специфические для указателя `shared_ptr`

<code>make_shared<T>(args)</code>	Возвращает указатель <code>shared_ptr</code> на динамически созданный объект типа T. Аргументы <code>args</code> используются для инициализации создаваемого объекта
<code>shared_ptr<T> p(q)</code>	p — копия <code>shared_ptr</code> q; инкремент счетчика q. Тип содержащегося в q указателя должен быть приводим к типу T* (см. раздел 4.11.2)
<code>p = q</code>	p и q — указатели <code>shared_ptr</code> , содержащие указатели, допускающие приведение друг к другу. Происходит декремент счетчика ссылок p и инкремент счетчика q; если счетчик указателя p достиг 0, память его объекта освобождается
<code>p.unique()</code>	Возвращает <code>true</code> , если <code>p.use_count()</code> равно единице, и значение <code>false</code> в противном случае
<code>p.use_count()</code>	Возвращает количество объектов, совместно использующих указатель p; может выполняться очень медленно, предназначена прежде всего для отладки

Функция `make_shared()`

Наиболее безопасный способ резервирования и использования динамической памяти подразумевает вызов библиотечной функции `make_shared()`. Она резервирует и инициализирует объект в динамической памяти, возвращая указатель типа `shared_ptr` на этот объект. Как и типы интеллектуальных указателей, функция `make_shared()` определена в заголовке `memory`.

При вызове функции `make_shared()` следует указать тип создаваемого объекта. Это подобно использованию шаблона класса — за именем функции следует указание типа в угловых скобках:

```

// указатель shared_ptr на объект типа int со
значением 42
shared_ptr<int> p3 = make_shared<int>(42);
// p4 указывает на строку со значением '9999999999'
shared_ptr<string> p4 = make_shared<string>(10,
'9');
// p5 указывает на объект типа int со значением по
// умолчанию (р. 3.3.1) 0
shared_ptr<int> p5 = make_shared<int>();

```

Подобно функции-члену `emplace()` последовательного контейнера (см. раздел 9.3.1), функция `make_shared()` использует свои аргументы для создания объекта заданного типа. Например, при вызове функции `make_shared<string>()` следует передать аргумент (аргументы), соответствующий одному из конструкторов типа `string`. Вызову функции `make_shared<int>()` можно передать любое значение, которое можно использовать для инициализации переменной типа `int`, и т.д. Если не передать аргументы, то объект инициализируется значением по умолчанию (см. раздел 3.3.1).

Для облегчения определения объекта, содержащего результат вызова функции `make_shared()`, обычно используют ключевое слово `auto` (см. раздел 2.5.2):

```

// p6 указывает на динамически созданный пустой
вектор vector<string>
auto p6 = make_shared<vector<string>>();

```

Копирование и присвоение указателей shared_ptr

При копировании и присвоении указателей `shared_ptr` каждый из них отслеживает количество других указателей `shared_ptr` на тот же объект:

```

auto p = make_shared<int>(42); // объект, на
который указывает p
                                // имеет только
одного владельца
auto q(p); // p и q указывают на тот же объект
// объект, на который указывают p и q, имеет двух
владельцев

```

С указателем `shared_ptr` связан счетчик, обычно называемый *счетчиком ссылок* (*reference count*). При копировании указателя

`shared_ptr` значение счетчика увеличивается. Например, значение связанного с указателем `shared_ptr` счетчика увеличивается, когда он используется для инициализации другого указателя `shared_ptr`, а также при использовании его в качестве правого операнда присвоения, или при передаче его функции (см. раздел 6.2.1), или при возвращении из функции по значению (см. раздел 6.3.2). Значение счетчика увеличивается при присвоении нового значения указателю `shared_ptr`, а когда он удаляется или когда локальный указатель `shared_ptr` выходит из области видимости (см. раздел 6.1.1), значение счетчика уменьшается.

Как только счетчик указателя `shared_ptr` достигает нуля, он автоматически освобождает объект, на который указывает:

```
auto r = make_shared<int>(42); // объект int, на
который указывает r,
// имеет одного
владельца
r = q; // присвоение r переводит этот указатель на
другой адрес
// приращение счетчика владельцев объекта, на
который указывает q
// уменьшение счетчика владельцев объекта, на
который указывает r
// объект, на который указывал r, не имеет более
владельцев;
// он освобождается автоматически
```

Здесь резервируется переменная типа `int`, а ее адрес сохраняется в указателе `r`. Затем указателю `r` присваивается новое значение. В данном случае `r` — единственный указатель типа `shared_ptr`, указывающий на этот объект. В результате присвоения `r = q` переменная `int` автоматически освобождается.



Будет ли использован счетчик или другая структура данных для отслеживания количества указателей на совместно используемый объект, зависит от реализации компилятора. Главное то, что класс отслеживает количество указателей `shared_ptr` на тот же объект и автоматически освобождает его в подходящий момент.

Указатель `shared_ptr` автоматически удаляет свои объекты...

Когда последний указатель `shared_ptr` на объект удаляется, его класс автоматически удаляет объект, на который он указывает. Для этого используется другая специальная функция-член — *деструктор* (destructor), аналогичная конструкторам, которые есть у каждого класса. Подобно тому, как конструктор контролирует инициализацию, деструктор контролирует происходящее при удалении объектов этого типа.

Деструкторы обычно освобождают ресурсы, зарезервированные объектом. Например, конструкторы класса `string` (как и другие его члены) резервируют память для содержания составляющих ее символов. Деструктор класса `string` освобождает эту память. Точно так же некоторые функции класса `vector` резервируют память для хранения элементов вектора. Деструктор класса `vector` удаляет эти элементы и освобождает используемую ими память.

Деструктор указателя `shared_ptr` осуществляет декремент счетчика ссылок объекта, на который он указывает. Когда счетчик достигает нуля, деструктор указателя `shared_ptr` удаляет объект, на который он указывает, и освобождает используемую им память.

...и автоматически освобождает их память

Тот факт, что класс `shared_ptr` автоматически освобождает динамические объекты, когда они больше не нужны, существенно облегчает использование динамической памяти. Рассмотрим, например, функцию, которая возвращает указатель `shared_ptr` на динамически созданный объект типа `Foo`, который может быть инициализирован аргументом типа `T`:

```
// функция factory() возвращает указатель
shared_ptr на динамически
// созданный объект
shared_ptr<Foo> factory( T arg ) {
    // обработать аргумент соответствующим образом
    // shared_ptr позаботится об освобождении этой
    // памяти
    return make_shared<Foo>( arg );
}
```

Функция `factory()` возвращает указатель `shared_ptr`, гарантирующий удаление созданного ею объекта в подходящий момент.

Например, следующая функция сохраняет указатель `shared_ptr`, возвращенный функцией `factory()`, в локальной переменной:

```
void use_factory( T arg) {  
    shared_ptr<Foo> p = factory( arg);  
    // использует p  
} // p выходит из области видимости; память, на  
которую он указывал,  
    // освобождается автоматически
```

Поскольку указатель `p` является локальным для функции `use_factory()`, он удаляется по ее завершении (см. раздел 6.1.1). Когда указатель `p` удаляется, осуществляется декремент его счетчика ссылок и проверка. В данном случае `p` — единственный указатель на объект в памяти, возвращенный функцией `factory()`. Поскольку указатель `p` выходит из области видимости, объект, на который он указывает, удаляется, а память, в которой он располагался, освобождается.

Память не будет освобождена, если на нее будет указывать любой другой указатель типа `shared_ptr`:

```
shared_ptr<Foo> use_factory( T arg) {  
    shared_ptr<Foo> p = factory( arg);  
    // использует p  
    return p; // при возвращении p счетчик ссылок  
    // увеличивается  
} // p выходит из области видимости; память, на  
которую он указывал,  
    // не освобождается
```

В этой версии функции `use_factory()` оператор `return` возвращает вызывающей стороне (см. раздел 6.3.2) копию указателя `p`. Копирование указателя `shared_ptr` добавляет единицу к счетчику ссылок этого объекта. Теперь, когда указатель `p` удаляется, останется другой владелец области памяти, на которую указывал указатель `p`. Класс `shared_ptr` гарантирует, что пока есть хоть один указатель `shared_ptr` на данную область памяти, она не будет освобождена.

Поскольку память не освобождается, пока не удаляется последний указатель `shared_ptr`, важно гарантировать, что ни одного указателя `shared_ptr` не остается после того, как необходимость в них отпадет. Если не удалить ненужный указатель `shared_ptr`, программа будет выполняться правильно, но может впустую тратить память. Одна из возможностей оставить указатели `shared_ptr` после употребления —

поместить их в контейнер, а затем переупорядочить его так, чтобы эти элементы оказались не нужны. Поэтому важно гарантировать удаление элементов с указателями `shared_ptr`, как только они больше не нужны.



Если указатели `shared_ptr` помещаются в контейнер, но впоследствии будут использованы лишь некоторые из них, а не все, то следует не забыть самостоятельно удалить остальные элементы.

Классы, ресурсы которых имеют динамическую продолжительность существования

Обычно динамическую память используют в следующих случаях.

1. Неизвестно необходимое количество объектов.
2. Неизвестен точный тип необходимых объектов.
3. Нельзя разрешать совместное использование данных несколькими объектами.

Классы контейнеров — хороший пример классов, использующих динамическую память, как в первом случае. Примеры второго рассматриваются в главе 15. В данном разделе определяется класс, использующий динамическую память для того, чтобы позволить нескольким объектам совместно использовать те же данные.

Использованные до сих пор классы резервировали ресурсы, которые существовали, только пока существовал объект. Например, каждому вектору принадлежат его собственные элементы. При копировании вектора элементы исходного вектора копировались в независимые элементы другого:

```
vector<string> v1; // пустой вектор
{ // новая область видимости
    vector<string> v2 = {"a", "an", "the"};
    v1 = v2; // копирует элементы из v2 в v1
} // v2 удаляется, что удаляет элементы v2
   // v1 содержит три элемента, являющихся копиями
элементов v2
```

Элементы вектора существуют, только пока существует сам вектор. Когда вектор удаляется, удаляются и его элементы.

Некоторые классы резервируют ресурсы, продолжительность

существования которых не зависит от первоначального объекта. Например, необходимо определить класс `Blob`, содержащий коллекцию элементов. В отличие от контейнеров, объекты класса `Blob` должны быть копиями друг друга и совместно использовать те же элементы. Таким образом, при копировании объекта класса `Blob` элементы копии должны ссылаться на те же элементы, что и оригинал.

Обычно, когда два объекта совместно используют те же данные, они не удаляются при удалении одного из объектов:

```
Blob<string> b1; // пустой Blob
{ // новая область видимости
    Blob<string> b2 = {"a", "an", "the"};
    b1 = b2; // b1 и b2 совместно используют те же
    элементы
} // b2 удаляется, но элементы b2 нет
// b1 указывает на элементы, первоначально
созданные в b2
```

В этом примере объекты `b1` и `b2` совместно используют те же элементы. Когда объект `b2` выходит из области видимости, эти элементы должны остаться, поскольку объект `b1` все еще использует их.

Основная причина использования динамической памяти в том, чтобы позволить нескольким объектам совместно использовать те же данные.

Определение класса `StrBlob`

В конечном счете класс `Blob` будет реализован как шаблон, но это только в разделе 16.1.2, а пока определим его версию, способную манипулировать только строками. Поэтому назовем данную версию этого класса `StrBlob`.

Простейший способ реализации нового типа коллекции подразумевает использование одного из библиотечных контейнеров. Это позволит библиотечному типу управлять собственно хранением элементов. В данном случае для хранения элементов будет использован класс `vector`.

Однако сам вектор не может храниться непосредственно в объекте `Blob`. Члены объекта удаляются при удалении самого объекта. Предположим, например, что объекты `b1` и `b2` класса `Blob` совместно используют тот же вектор. Если бы вектор хранился в одном из этих объектов, скажем в `b2`, то, как только объект `b2` выйдет из области видимости, элементы вектора перестанут существовать. Чтобы гарантировать продолжение существования элементов, будем хранить

вектор в динамической памяти.

Для реализации совместного использования снабдим каждый объект класса `StrBlob` указателем `shared_ptr` на вектор в динамической памяти. Указатель-член `shared_ptr` будет следить за количеством объектов класса `StrBlob`, совместно использующих тот же самый вектор, и удалит его, когда будет удален последний объект класса `StrBlob`.

Осталось решить, какие функции будет предоставлять создаваемый класс. Реализуем пока небольшое подмножество функций вектора. Изменим также функции обращения к элементам (включая `front()` и `back()`): в данном классе при попытке доступа к не существующим элементам они будут передавать исключения.

У класса будет стандартный конструктор и конструктор с параметром типа `initializer_list<string>` (см. раздел 6.2.6). Этот конструктор будет получать список инициализаторов в скобках.

```
class StrBlob {
public:
    typedef std::vector<std::string>::size_type
size_type;
    StrBlob();
    StrBlob(std::initializer_list<std::string> il);
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
    // добавление и удаление элементов
    void push_back(const std::string &t) { data-
>push_back(t); }
    void pop_back();
    // доступ к элементам
    std::string& front();
    std::string& back();
private:
    std::shared_ptr<std::vector<std::string>> data;
    // передать сообщение при недопустимости data[i]
    void check(size_type i, const std::string &msg)
const;
};
```

В классе будут реализованы функции-члены `size()`, `empty()` и `push_back()`, которые передают свою работу через указатель `data` внутреннему вектору. Например, функция `size()` класса `StrBlob`

вызывает функцию `data->size()` и т.д.

Конструкторы класса StrBlob

Для инициализации своей переменной-члена `data` указателем на динамически созданный вектор каждый конструктор использует собственный список инициализации (см. раздел 7.1.4). Стандартный конструктор резервирует пустой вектор:

```
StrBlob::StrBlob():  
data( make_shared<vector<string>>() ) { }  
StrBlob::StrBlob( initializer_list<string> il):  
data( make_shared<vector<string>>( il ) ) { }
```

Конструктор, получающий тип `initializer_list`, передает свой параметр для соответствующего конструктора класса `vector` (см. раздел 2.2.1). Этот конструктор инициализирует элементы вектора копиями значений из списка.

Функции-члены доступа к элементам

Функции `pop_back()`, `front()` и `back()` обращаются к соответствующим функциям-членам вектора. Эти функции должны проверять существование элементов прежде, чем попытаться получить доступ к ним. Поскольку несколько функций-членов должны осуществлять ту же проверку, снабдим класс закрытой вспомогательной функцией `check()`, проверяющей принадлежность заданного индекса диапазону. Кроме индекса, функция `check()` получает аргумент типа `string`, передаваемый обработчику исключений. Стока описывает то, что пошло не так, как надо:

```
void StrBlob::check( size_type i, const string &msg)  
const {  
    if ( i >= data->size() )  
        throw out_of_range( msg );  
}
```

Функция `pop_back()` и функции-члены доступа к элементам сначала вызывают функцию `check()`. Если проверка успешна, эти функции-члены передают свою работу соответствующим функциям вектора:

```
strings StrBlob::front() {  
    // если вектор пуст, функция check() передаст  
    // следующее  
    check( 0, "front on empty StrBlob" );
```

```

    return data->front();
}
strings StrBlob::back() {
    check(0, "back on empty StrBlob");
    return data->back();
}
void StrBlob::pop_back() {
    check(0, "pop_back on empty StrBlob");
    data->pop_back();
}

```

Функции-члены `front()` и `back()` должны быть перегружены для констант (см. раздел 7.3.2). Определение этих версий остается в качестве самостоятельного упражнения.

Копирование, присвоение и удаление объектов класса `StrBlob`

Подобно классу `Sales_data`, класс `StrBlob` использует стандартные версии функций копирования, присвоения и удаления объектов (см. раздел 7.1.5). По умолчанию эти функции копируют, присваивают и удаляют переменные-члены класса. У класса `StrBlob` есть только одна переменная-член — указатель `shared_ptr`. Поэтому при копировании, присвоении и удалении объекта класса `StrBlob` его переменная-член `shared_ptr` будет скопирована, присвоена или удалена.

Как уже упоминалось выше, копирование указателя `shared_ptr` приводит к инкременту его счетчика ссылок; присвоение одного указателя `shared_ptr` другому приводит к инкременту счетчика правого операнда и декременту счетчика левого; удаление указателя `shared_ptr` приводит к декременту его счетчика. Если значение счетчика указателя `shared_ptr` доходит до нуля, объект, на который он указывает, удаляется автоматически. Таким образом, вектор, созданный конструкторами класса `StrBlob`, будет автоматически удален при удалении последнего объекта класса `StrBlob`, указывающего на этот вектор.

Упражнения раздела 12.1.1

Упражнение 12.1. Сколько элементов будут иметь объекты `b1` и `b2` в конце этого кода?

```

StrBlob b1;
StrBlob b2 = {"a", "an", "the"};

```

```
b1 = b2;  
b2.push_back("about");  
}
```

Упражнение 12.2. Напишите собственную версию класса `StrBlob`, включающую константные версии функций `front()` и `back()`.

Упражнение 12.3. Нуждается ли этот класс в константных версиях функций `push_back()` и `pop_back()`? Если они нужны, добавьте их. В противном случае объясните, почему они не нужны?

Упражнение 12.4. В функции `check()` нет проверки того, что параметр `i` больше нуля. Почему эта проверка не нужна?

Упражнение 12.5. Конструктор, получающий тип `initializer_list`, не был объявлен как `explicit` (см. раздел 7.5.4). Обсудите преимущества и недостатки этого выбора.

12.1.2. Непосредственное управление памятью

Язык определяет два оператора, позволяющие резервировать и освобождать области в динамической памяти. Оператор `new` резервирует память, а оператор `delete` освобождает память, зарезервированную оператором `new`.

По причинам, которые станут ясны позже, использование этих операторов для управления памятью существенно более подвержено ошибкам, чем использование интеллектуальных указателей. Кроме того, классы, самостоятельно управляющие памятью (в отличие от таковых, использующих интеллектуальные указатели), не могут полагаться на стандартные определения тех их членов, которые копируют, присваивают и удаляют объекты класса (см. раздел 7.1.4). В результате программы, использующие интеллектуальные указатели, вероятно, будет проще написать и отлаживать.



ВНИМАНИЕ

Пока не пройдена глава 13, будем использовать классы, резервирующие динамическую память, *только если для управления ею используются интеллектуальные указатели*.

Использование оператора `new` для динамического резервирования и инициализации объектов

Созданные в динамической памяти объекты не имеют имен, поэтому *оператор `new` не предполагает никаких способов именования резервируемых объектов*. Вместо этого оператор `new` возвращает указатель на зарезервированный объект:

```
int *pi = new int; // pi указывает на динамически созданный,
```

```
                    // безымянный,
```

```
                    // неинициализированный объект
```

типа `int`

Это выражение `new` создает в динамической памяти объект типа `int` и возвращает указатель на него.

По умолчанию создаваемые в динамической памяти объекты инициализируются значением по умолчанию (см. раздел 2.2.1). Это значит,

что у объектов встроенного или составного типа будет неопределенное значение, а объекты типа класса инициализируются их стандартным конструктором:

```
string *ps = new string; // инициализируется пустой строкой
int *pi = new int; // pi указывает на неинициализированный int
```



Динамически созданный объект можно инициализировать, используя прямую инициализацию (см. раздел 3.2.1). Можно применить традиционный конструктор (используя круглые скобки), а по новому стандарту можно также использовать списочную инициализацию (с фигурными скобками):

```
int *pi = new int(1024); // pi указывает на объект со значением 1024
string *ps = new string(10, '9'); // *ps =
"9999999999"
// вектор на десять элементов со значениями от 0 до 9
vector<int> *pv = new vector<int>
{0,1,2,3,4,5,6,7,8,9};
```

Динамически созданный объект можно также инициализировать значением по умолчанию (см. раздел 3.3.1), сопроводив имя типа парой пустых круглых скобок:

```
string *ps1 = new string; // инициализация по умолчанию пустой строкой
string *ps = new string(); // инициализация значением по умолчанию // (пустой строкой)
int *pi1 = new int; // инициализация по умолчанию; // значение *pi1 не определено
int *pi2 = new int(); // инициализация значением по умолчанию 0; // *pi2 = 0
```

Для типов классов (таких как `string`), определяющих собственные

конструкторы (см. раздел 7.1.4), запрос инициализации значением по умолчанию не имеет последствий; независимо от формы, объект инициализируется стандартным конструктором. Различие существенно в случае встроенных типов: инициализация объекта встроенного типа значением по умолчанию присваивает ему вполне конкретное значение, а инициализация по умолчанию — нет. Точно так же полагающиеся на синтезируемый стандартный конструктор члены класса встроенного типа также не будут не инициализированы, если эти члены не будут инициализированы в теле класса (см. раздел 7.1.4).

Рекомендуем

По тем же причинам, по которым обычно инициализируют переменные, имеет смысл инициализировать и динамически созданные объекты.



Когда предоставляется инициализатор в круглых скобках, для вывода типа объекта, который предстоит зарезервировать для этого инициализатора, можно использовать ключевое слово `auto` (см. раздел 2.5.2). Но, поскольку компилятор использует тип инициализатора для вывода резервируемого типа, ключевое слово `auto` можно использовать только с одиночным инициализатором в круглых скобках:

```
auto p1 = new auto( obj ); // p указывает на объект
                           // этого объекта
инициализируется значением obj
auto p2 = new auto{ a, b, c }; // ошибка: для
инициализатора нужно
                           // использовать круглые
скобки
```

Тип `p1` — это указатель на автоматически выведененный тип `obj`. Если `obj` имеет тип `int`, то тип `p1` — `int*`; если `obj` имеет тип `string`, то тип `p1` — `string*` и т.д. Вновь созданный объект инициализируется значением объекта `obj`.

Динамически созданные константные объекты

Для резервирования константных объектов вполне допустимо

использовать оператор new:

```
// зарезервировать и инициализировать
const int const int *pci = new const int(1024);
// зарезервировать и инициализировать значением по
умолчанию
const string const string *pcs = new const string;
```

Подобно любым другим константным объектам, динамически созданный константный объект следует инициализировать. Динамический константный объект типа класса, определяющего стандартный конструктор (см. раздел 7.1.4), можно инициализировать неявно. Объекты других типов следует инициализировать явно. Поскольку динамически зарезервированный объект является константой, возвращенный оператором new указатель является указателем на константу (см. раздел 2.4.2).

Исчерпание памяти

Хотя современные машины имеют огромный объем памяти, всегда существует вероятность исчерпания динамической памяти. Как только программа использует всю доступную ей память, выражения с оператором new будут терпеть неудачу. По умолчанию, если оператор new неспособен зарезервировать требуемый объем памяти, он передает исключение типа bad_alloc (см. раздел 5.6). Используя иную форму оператора new, можно воспрепятствовать передаче исключения:

```
// при неудаче оператор new возвращает нулевой
указатель
int *p1 = new int; // при неудаче оператор new
передает
                                // исключение std::bad_alloc
int *p2 = new (nothrow) int; // при неудаче
оператор new возвращает
                                // нулевой указатель
```

По причинам, рассматриваемым в разделе 19.1.2, эта форма оператора new упоминается как *размещающий оператор new* (placement new). Выражение размещающего оператора new позволяет передать дополнительные аргументы. В данном случае передается определенный библиотекой объект nothrow. Передача объекта nothrow оператору new указывает, что он не должен передавать исключения. Если эта форма оператора new окажется неспособна зарезервировать требуемый объем памяти, она возвратит нулевой указатель. Типы bad_alloc и nothrow

определенены в заголовке `new`.

Освобождение динамической памяти

Чтобы предотвратить исчерпание памяти, по завершении использования ее следует возвратить операционной системе. Для этого используется оператор `delete`, получающий указатель на освобождаемый объект:

```
delete p; // p должен быть указателем на  
динамически созданный объект  
// или нулевым указателем
```

Подобно оператору `new`, оператор `delete` выполняет два действия: удаляет объект, на который указывает переданный ему указатель, и освобождает соответствующую область памяти.

Значения указателя и оператор delete

Передаваемый оператору `delete` указатель должен либо указывать на динамически созданный объект, либо быть нулевым указателем (см. раздел 2.3.2). Результат удаления указателя на область памяти, зарезервированную не оператором `new`, или повторного удаления значения того же указателя непредсказуем:

```
int i, *pi1 = &i, *pi2 = nullptr;  
double *pd = new double(33), *pd2 = pd;  
delete i; // ошибка: i - не указатель  
delete pi1; // непредсказуемо: pi1 - локальный  
delete pd; // ok  
delete pd2; // непредсказуемо: память, на которую  
указывает pd2,  
// уже освобождена  
delete pi2; // ok: освобождение нулевого указателя  
всегда допустимо
```

Компилятор сообщает об ошибке оператора `delete i`, поскольку знает, что `i` — не указатель. Ошибки, связанные с выполнением оператора `delete` для указателей `pi1` и `pd2`, коварней: обычно компиляторы неспособны выяснить, указывает ли указатель на объект, созданный статически или динамически. Точно так же компилятор не может установить, была ли уже освобождена память, на которую указывает указатель. Большинство компиляторов примет такие выражения `delete`, несмотря на их ошибочность.

Хотя значение константного объекта не может быть изменено, сам объект вполне может быть удален. Подобно любым динамическим объектам, константный динамический объект освобождается выполнением оператора `delete` для указателя, указывающего на этот объект:

```
const int *pci = new const int(1024);  
delete pci; // ok: удаляет константный объект
```

*Динамически созданные объекты существуют до тех пор,
пока не будут освобождены*

Как упоминалось в разделе 12.1.1, управляемая указателем `shared_ptr` память автоматически освобождается при удалении последнего указателя `shared_ptr`. Динамический объект, управляемый указателем встроенного типа, существует до тех пор, пока к областям памяти, управляемой при помощи указателей встроенных типов, не будет удален явно.

Функции, возвращающие обычные (а не интеллектуальные) указатели на области динамической памяти, возлагают ответственность за их удаление на вызывающую сторону:

```
// возвращает указатель на динамически созданный  
объект  
Foo* factory(T arg) {  
    // обработать аргумент соответственно  
    return new Foo(arg); // за освобождение этой  
памяти отвечает  
    // вызывающая сторона  
}
```

Подобно прежней версии функции `factory()` (см. раздел 12.1.1), эта версия резервирует объект, но не удаляет его. Ответственность за освобождение памяти динамического объекта, когда он станет больше не нужен, несет вызывающая сторона функции `factory()`. К сожалению, вызывающая сторона слишком часто забывает сделать это:

```
void use_factory(T arg) {  
    Foo *p = factory(arg);  
    // использовать p, но не удалить его  
} // p выходит из области видимости, но память,  
// на которую он указывает, не освобождается!
```

Здесь функция `use_factory()` вызывает функцию `factory()` резервирующую новый объект типа `Foo`. Когда функция `use_factory()`

завершает работу, локальная переменная `p` удаляется. Эта переменная — встроенный указатель, а не интеллектуальный.

В отличие от классов, при удалении объектов встроенного типа не происходит ничего. В частности, когда указатель выходит из области видимости, с объектом, на который он указывает, ничего не происходит. Если этот указатель указывает на динамическую память, она не освобождается автоматически.



Динамическая память, управляемая при помощи встроенных (а не интеллектуальных) указателей, продолжает существование, пока не будет освобождена явно.

В этом примере указатель `p` был единственным указателем на область памяти, зарезервированную функцией `factory()`. По завершении функции `use_factory()` у программы больше нет никакого способа освободить эту память. Согласно общей логике программирования, следует исправить эту ошибку и напомнить о необходимости освобождения памяти в функции `use_factory()`:

```
void use_factory( T arg ) {  
    Foo *p = factory( arg );  
    // использование p  
    delete p; // не забыть освободить память сейчас,  
когда  
    // она больше не нужна  
}
```

Если созданный функцией `use_factory()` объект должен использовать другой код, то эту функцию следует изменить так, чтобы она возвращала указатель на зарезервированную ею память:

```
Foo* use_factory( T arg ) {  
    Foo *p = factory( arg );  
    // использование p  
    return p; // освободить память должна вызывающая  
сторона  
}
```

Внимание! Управление динамической памятью подвержено ошибкам

Есть три общеизвестных проблемы, связанных с использованием операторов `new` и `delete` при управлении динамической памятью:

1. Память забыли освободить. Когда динамическая память не освобождается, это называется "утечка памяти", поскольку она уже не возвращается в пул динамической памяти. Проверка утечек памяти очень трудна, поскольку она обычно не проявляется, пока приложение, проработав достаточно долго, фактически не исчерпает память.

2. Объект использован после удаления. Иногда эта ошибка обнаруживается при создании нулевого указателя после удаления.

3. Повторное освобождение той же памяти. Эта ошибка может произойти в случае, когда два указателя указывают на тот же динамически созданный объект. Если оператор `delete` применен к одному из указателей, то память объекта возвращается в пул динамической памяти. Если впоследствии применить оператор `delete` ко второму указателю, то динамическая память может быть нарушена.

Допустить эти ошибки значительно проще, чем потом найти и исправить.



Избежать *всех* этих проблем при использовании исключительно интеллектуальных указателей не получится. Интеллектуальный указатель способен позаботиться об удалении памяти *только* тогда, когда не останется других интеллектуальных указателей на эту область памяти.

Переустановка значения указателя после удаления...

Когда указатель удаляется, он становится недопустимым. Но, даже став недопустимым, на многих машинах он продолжает содержать адрес уже освобожденной области динамической памяти. После освобождения области памяти указатель на нее становится *потерянным указателем* (*dangling pointer*). Потерянный указатель указывает на ту область памяти, которая когда-то содержала объект, но больше не содержит.

Потерянным указателям присущи все проблемы неинициализированных указателей (см. раздел 2.3.2). Проблем с потерянными указателями можно избежать, освободив связанную с ними память непосредственно перед выходом из области видимости самого указателя. Так не появится шанса использовать указатель уже после того, как связанная с ним память будет освобождена. Если указатель

необходимо сохранить, то после применения оператора `delete` ему можно присвоить значение `nullptr`. Это непосредственно свидетельствует о том, что указатель не указывает на объект.

...обеспечивает лишь частичную защиту

Фундаментальная проблема с динамической памятью в том, что может быть несколько указателей на ту же область памяти. Переустановка значения указателя при освобождении памяти позволяет проверять допустимость данного конкретного указателя, но никак не влияет на все остальные указатели, все еще указывающие на уже освобожденную область памяти. Рассмотрим пример:

```
int *p( new int( 42 ) ); // p указывает на динамическую  
память  
auto q = p; // p и q указывают на ту же  
область памяти  
delete p; // делает недопустимыми p и q  
p = nullptr; // указывает, что указатель p больше  
не связан с объектом
```

Здесь указатели `p` и `q` указывают на тот же динамически созданный объект. Удалим этот объект и присвоим указателю `p` значение `nullptr`, засвидетельствовав, что он больше не указывает на объект. Однако переустановка значения указателя `p` никак не влияет на указатель `q`, который стал недопустимым после освобождения памяти, на которую указывал указатель `p` (и указатель `q`!). В реальных системах поиск всех указателей на ту же область памяти зачастую на удивление труден.

Упражнения раздела 12.1.2

Упражнение 12.6. Напишите функцию, которая возвращает динамически созданный вектор целых чисел. Передайте этот вектор другой функции, которая читает значения его элементов со стандартного устройства ввода. Передайте вектор другой функции, выводящей прочитанные ранее значения. Не забудьте удалить вектор в подходящий момент.

Упражнение 12.7. Переделайте предыдущее упражнение, используя на сей раз указатель `shared_ptr`.

Упражнение 12.8. Объясните, все ли правильно в следующей функции:

```
bool b( ) {  
    int* p = new int;  
    // ...
```

```
    return p;
}
```

Упражнение 12.9. Объясните, что происходит в следующем коде:

```
int *q = new int( 42 ), *r = new int( 100 );
r = q;
auto q2      = make_shared<int>( 42 ),      r2      =
make_shared<int>( 100 );
r2 = q2;
```

12.1.3. Использование указателя **shared_ptr** с оператором **new**

Как уже упоминалось, если не инициализировать интеллектуальный указатель, он инициализируется как нулевой. Как свидетельствует табл. 12.3, интеллектуальный указатель можно также инициализировать указателем, возвращенным оператором `new`:

```
shared_ptr<double> p1; // shared_ptr может
указывать на double
shared_ptr<int> p2( new int( 42 ) ); // p2 указывает на
int со значением 42
```

Конструкторы интеллектуального указателя, получающие указатели, являются явными (см. раздел 7.5.4). Следовательно, нельзя неявно преобразовать встроенный указатель в интеллектуальный; для инициализации интеллектуального указателя придется использовать прямую форму инициализации (см. раздел 3.2.1):

```
shared_ptr<int> p1 = new int( 1024 ); // ошибка:
нужна
// прямая
инициализация
shared_ptr<int> p2( new int( 1024 ) ); // ok:
использует
// прямую
инициализацию
```

Таблица 12.3. Другие способы определения и изменения указателя **shared_ptr**

<code>shared_ptr<T> p(q)</code>	Указатель <code>p</code> управляет объектом, на который указывает указатель встроенного типа <code>q</code> ; указатель <code>q</code> должен указывать на область памяти, зарезервированную оператором <code>new</code> , а его тип должен быть
---	--

	преобразуем в тип T^*
<code>shared_ptr<T> p(u)</code>	Указатель p учитывает собственность указателя u типа <code>unique_ptr</code> ; указатель u становится нулевым
<code>shared_ptr<T> p(q, d)</code>	Указатель p учитывает собственность объекта, на который указывает встроенный указатель q . Тип указателя q должен быть преобразуем в тип T^* (см. раздел 4.11.2). Для освобождения q указатель p будет использовать вызываемый объект d (см. раздел 10.3.2) вместо оператора <code>delete</code>
<code>shared_ptr<T> p(p2, d)</code>	Указатель p — это копия указателя $p2$ типа <code>shared_ptr</code> , как описано в табл. 12.2, за исключением того, что указатель p использует вызываемый объект d вместо оператора <code>delete</code>
<code>p.reset()</code> <code>p.reset(q)</code> <code>p.reset(q, d)</code>	Если p единственный указатель <code>shared_ptr</code> на объект, функция <code>reset()</code> освободит существующий объект p . Если передан необязательный встроенный указатель q , то p будет указывать на q , в противном случае p станет нулевым. Если предоставлен вызываемый объект d , то он будет вызван для освобождения указателя q , в противном случае используется оператор <code>delete</code>

Инициализация указателя $p1$ неявно требует, чтобы компилятор создал указатель типа `shared_ptr` из указателя `int*`, возвращенного оператором `new`. Поскольку нельзя неявно преобразовать обычный указатель в интеллектуальный, такая инициализация ошибочна. По той же причине функция, возвращающая указатель `shared_ptr`, не может неявно преобразовать простой указатель в своем операторе `return`:

```
shared_ptr<int> clone( int p ) {
    return new int( p );      // ошибка: неявное
    преобразование
                                // в shared_ptr<int>
}
```

Следует явно связать указатель `shared_ptr` с указателем, который предстоит возвратить:

```
shared_ptr<int> clone ( int p ) {
    // ok: явное создание shared_ptr<int> из int*
    return shared_ptr<int>( new int( p ) );
}
```

По умолчанию указатель, используемый для инициализации интеллектуального указателя, должен указывать на область динамической памяти, поскольку по умолчанию интеллектуальные указатели используют оператор `delete` для освобождения связанного с ним объекта. Интеллектуальные указатели можно связать с указателями на другие виды

ресурсов. Но для этого необходимо предоставить собственную функцию, используемую вместо оператора `delete`. Предоставление собственного кода удаления рассматривается в разделе 12.1.4.



Не смешивайте обычные указатели с интеллектуальными

Указатель `shared_ptr` может координировать удаление только с другими указателями `shared_ptr`, которые являются его копиями. Действительно, этот факт — одна из причин, по которой рекомендуется использовать функцию `make_shared()`, а не оператор `new`. Это связывает указатель `shared_ptr` с объектом одновременно с его резервированием. При этом нет никакого способа по неосторожности связать ту же область памяти с несколькими независимо созданными указателями `shared_ptr`.

Рассмотрим следующую функцию,工作的 с указателем `shared_ptr`:

```
// ptr создается и инициализируется при вызове
process()
void process( shared_ptr<int> ptr) {
    // использование ptr
} // ptr выходит из области видимости и удаляется
```

Параметр функции `process()` передается по значению, поэтому аргумент копируется в параметр `ptr`. Копирование указателя `shared_ptr` осуществляет инкремент его счетчика ссылок. Таким образом, в функции `process()` значение счетчика не меньше 2. По завершении функции `process()` осуществляется декремент счетчика ссылок указателя `ptr`, но он не может достигнуть нуля. Поэтому, когда локальная переменная `ptr` удаляется, память, на которую она указывает, не освобождается.

Правильный способ использования этой функции подразумевает передачу ей указателя `shared_ptr`:

```
shared_ptr<int> p( new int ( 42 )); // счетчик ссылок
= 1
process( p ); // копирование p увеличивает счетчик;
               // в функции process() счетчик = 2
int i = *p; // ok: счетчик ссылок = 1
```

Хотя функции `process()` нельзя передать встроенный указатель, ей

можно передать временный указатель `shared_ptr`, явно созданный из встроенного указателя. Но это, вероятно, будет ошибкой:

```
int *x( new int(1024)); // опасно: x - обычный
указатель, а
                                         // не интеллектуальный
process(x);
// ошибка: нельзя преобразовать int* в
shared_ptr<int>
process(shared_ptr<int>(x)); // допустимо, но
память будет освобождена!
int j = *x; // непредсказуемо: x - потерянный
указатель!
```

В этом вызове функции `process()` передан временный указатель `shared_ptr`. Этот временный указатель удаляется, когда завершается выражение, в котором присутствует вызов. Удаление временного объекта приводит к декременту счетчика ссылок, доводя его до нуля. Память, на которую указывает временный указатель, освобождается при удалении временного указателя.

Но указатель `x` продолжает указывать на эту (освобожденную) область памяти; теперь `x` — потерянный указатель. Результат попытки использования значения, на которое указывает указатель `x`, непредсказуем.

При связывании указателя `shared_ptr` с простым указателем ответственность за эту память передается указателю `shared_ptr`. Как только ответственность за область памяти встроенного указателя передается указателю `shared_ptr`, больше нельзя использовать встроенный указатель для доступа к памяти, на которую теперь указывает указатель `shared_ptr`.



Опасно использовать встроенный указатель для доступа к объекту, принадлежащему интеллектуальному указателю, поскольку нельзя быть уверенным в том, что этот объект еще не удален.

Другие операции с указателем `shared_ptr`

Класс `shared_ptr` предоставляет также несколько других операций, перечисленных в табл. 12.2 и табл. 12.3. Чтобы присвоить новый указатель

указателю `shared_ptr`, можно использовать функцию `reset()`:

```
p = new int(1024); // нельзя присвоить обычный
указатель
                                // указателю shared_ptr
p.reset(new int(1024)); // ok: p указывает на новый
объект
```

Подобно оператору присвоения, функция `reset()` модифицирует счетчики ссылок, а если нужно, удаляет объект, на который указывает указатель `p`. Функцию-член `reset()` зачастую используют вместе с функцией `unique()` для контроля совместного использования объекта несколькими указателями `shared_ptr`. Прежде чем изменять базовый объект, проверяем, является ли владелец единственным. В противном случае перед изменением создается новая копия:

```
if (!p.unique())
    p.reset(new string(*p)); // владелец не один;
резервируем новую копию
*p += newVal; // теперь, когда известно, что
указатель единственный,
                                // можно изменить объект
```

Упражнения раздела 12.1.3

Упражнение 12.10. Укажите, правилен ли следующий вызов функции `process()`, определенной в текущем разделе. В противном случае укажите, как его исправить?

```
shared_ptr<int> p(new int(42));
process(shared_ptr<int>(p));
```

Упражнение 12.11. Что будет, если вызвать функцию `process()` следующим образом?

```
process(shared_ptr<int>(p.get()));
```

Упражнение 12.12. Используя объявления указателей `p` и `sp`, объясните каждый из следующих вызовов функции `process()`. Если вызов корректен, объясните, что он делает. Если вызов некорректен, объясните почему:

```
auto p = new int();
auto sp = make_shared<int>();
(a) process(sp);
(b) process(new int());
(c) process(p);
```

```
( d) process( shared_ptr<int>( p ) );
```

Упражнение 12.13. Что будет при выполнении следующего кода?

```
auto sp = make_shared<int>();  
auto p = sp.get();  
delete p;
```



12.1.4. Интеллектуальные указатели и исключения

В разделе 5.6.2 упоминалось, что программы, использующие обработку исключений для продолжения работы после того, как произошло исключение, нуждаются в способе правильного освобождения ресурсов в случае исключения. Самый простой из них подразумевает использование интеллектуальных указателей.

При использовании интеллектуального указателя его класс гарантирует освобождение памяти, когда в ней больше нет необходимости, даже при преждевременном выходе из блока:

```
void f() {  
    shared_ptr<int>     sp( new     int( 42 ) );           //  
    зарезервировать новый объект  
    // код, передающий исключение, не обрабатываемое в  
    функции f()  
} // shared_ptr освобождает память автоматически по  
завершении функции
```

При выходе из функции, обычном или в связи с исключением, удаляются все ее локальные объекты. В данном случае указатель `sp` имеет тип `shared_ptr`, поэтому при удалении проверяется его счетчик ссылок. В данном случае `sp` — единственный указатель на контролируемую им область памяти, поэтому она освобождается в ходе удаления указателя `sp`.

Память, контролируемая непосредственно, напротив, не освобождается автоматически, когда происходит исключение. Если для управления памятью используются встроенные указатели и исключение происходит после оператора `new`, но перед оператором `delete`, то контролируемая память не будет освобождена:

```
void f() {  
    int *ip = new int( 42 );           // динамически  
    зарезервировать новый объект
```

```
// код, передающий исключение, не обрабатываемое в
функции f()
    delete ip; // освобождает память перед выходом
}
```

Если исключение происходит между операторами `new` и `delete` и не обрабатывается в функции `f()`, то освободить эту память никак не получится. Вне функции `f()` нет указателя на эту память, поэтому нет никакого способа освободить ее.



Интеллектуальные указатели и классы без деструкторов

Большинство классов языка C++, включая все библиотечные классы, определяют деструкторы (см. раздел 12.1.1), заботящиеся об удалении используемых объектом ресурсов. Но не все классы таковы. В частности, классы, разработанные для использования и в языке C, и в языке C++, обычно требуют от пользователя явного освобождения всех используемых ресурсов.

Классы, которые резервируют ресурсы, но не определяют деструкторы для их освобождения, подвержены тем же ошибкам, которые возникают при самостоятельном использовании динамической памяти. Довольно просто забыть освободить ресурс. Аналогично, если произойдет исключение после резервирования ресурса, но до его освобождения, программа потеряет его.

Для управления классами без деструкторов зачастую можно использовать те же подходы, что и для управления динамической памятью. Предположим, например, что используется сетевая библиотека, применимая как в языке C, так и в C++. Использующая эту библиотеку программа могла бы содержать такой код:

```
struct destination; // представляет то, с чем
установлено соединение
struct connection; // информация для использования
соединения
connection connect(destination*); // открывает
соединение
void disconnect(connection); // закрывает
данное соединение
void f(destination &d /* другие параметры */) {
    // получить соединение; не забыть закрывать по
```

завершении

```
connection c = connect( &d ); // использовать
соединение
// если забыть вызывать функцию disconnect() перед
выходом из
// функции f(), то уже не будет никакого способа
закрыть соединение
}
```

Если бы у структуры `connection` был деструктор, то по завершении функции `f()` он закрыл бы соединение автоматически. Однако у нее нет деструктора. Эта проблема почти идентична проблеме предыдущей программы, использовавшей указатель `shared_ptr`, чтобы избежать утечек памяти. Здесь также можно использовать указатель `shared_ptr` для гарантии правильности закрытия соединения.



Использование собственного кода удаления

По умолчанию указатели `shared_ptr` подразумевали, что они указывают на динамическую память. Следовательно, когда указатель `shared_ptr` удаляется, он по умолчанию выполняет оператор `delete` для содержащегося в нем указателя. Чтобы использовать указатель `shared_ptr` для управления соединением `connection`, следует сначала определить функцию, используемую вместо оператора `delete`. Должна быть возможность вызова этой *функции удаления* (*deleter*) с указателем, хранимым в указателе `shared_ptr`. В данном случае функция удаления должна получать один аргумент типа `connection*`:

```
void end_connection( connection * p )
{
    disconnect( *p );
}
```

При создании указателя `shared_ptr` можно передать необязательный аргумент, указывающий на функцию удаления (см. раздел 6.7):

```
void f( destination &d /* другие параметры */ ) {
    connection c = connect( &d );
    shared_ptr<connection> p( &c, end_connection );
    // использовать соединение
    // при выходе из функции f(), даже в случае
    исключения, соединение
    // будет закрыто правильно
}
```

}

При удалении указателя `p` для хранимого в нем указателя вместо оператора `delete` будет вызвана функция `end_connection()`. Функция `end_connection()`, в свою очередь, вызовет функцию `disconnect()`, гарантируя таким образом закрытие соединения. При нормальном выходе из функции `f()` указатель `p` будет удален в ходе процедуры выхода. Кроме того, указатель `p` будет также удален, а соединение закрыто, если произойдет исключение.

Внимание! Проблемы интеллектуального указателя

Интеллектуальные указатели могут обеспечить безопасность и удобство работы с динамически созданной памятью только при правильном использовании. Для этого следует придерживаться ряда соглашений.

- Не используйте значение того же встроенного указателя для инициализации (переустановки) нескольких интеллектуальных указателей.
- Не используйте оператор `delete` для указателя, возвращенного функцией `get()`.
- Не используйте функцию `get()` для инициализации или переустановки другого интеллектуального указателя.
- Используя указатель, возвращенный функцией `get()`, помните, что указатель станет недопустимым после удаления последнего соответствующего интеллектуального указателя.
- Если интеллектуальный указатель используется для управления ресурсом, отличным от области динамической памяти, зарезервированной оператором `new`, не забывайте использовать функцию удаления (раздел 12.1.4 и раздел 12.1.5).

Упражнения раздела 12.1.4

Упражнение 12.14. Напишите собственную версию функции, использующую указатель `shared_ptr` для управления соединением.

Упражнение 12.15. Перепишите первое упражнение так, чтобы использовать лямбда-выражение (см. раздел 10.3.2) вместо функции `end_connection()`.

12.1.5. Класс `unique_ptr`

Указатель `unique_ptr` "владеет" объектом, на который он указывает. В отличие от указателя `shared_ptr`, только один указатель `unique_ptr` может одновременно указывать на данный объект. Объект, на который указывает указатель `unique_ptr`, удаляется при удалении указателя. Список функций, специфических для указателя `unique_ptr`, приведен в табл. 12.4. Функции, общие для обоих указателей, приведены в табл. 12.1.

В отличие от указателя `shared_ptr`, нет никакой библиотечной функции, подобной функции `make_shared()`, которая возвращала бы указатель `unique_ptr`. Вместо этого определяемый указатель `unique_ptr` связывается с указателем, возвращенным оператором `new`. Подобно указателю `shared_ptr`, можно использовать прямую форму инициализации:

```
unique_ptr<double> p1; // указатель unique_ptr на тип double
```

```
unique_ptr<int> p2( new int( 42 ) ); // p2 указывает на int со значением 42
```

Таблица 12.4. Функции указателя `unique_ptr` (см. также табл. 12.1)

<code>unique_ptr<T> u1 unique_ptr<T, D> u2</code>	Обнуляет указатель <code>unique_ptr</code> , способный указывать на объект типа <code>T</code> . Указатель <code>u1</code> используется для освобождения своего указателя оператором <code>delete</code> ; а указатель <code>u2</code> — вызываемый объект типа <code>D</code>
<code>unique_ptr<T, D> u(d)</code>	Обнуляет указатель <code>unique_ptr</code> , указывающий на объекты типа <code>T</code> . Использует вызываемый объект <code>d</code> типа <code>D</code> вместо оператора <code>delete</code>
<code>u = nullptr</code>	Удаляет объект, на который указывает указатель <code>u</code> ; обнуляет указатель <code>u</code>
<code>u. release()</code>	Прекращает контроль содержимого указателя <code>u</code> ; возвращает содержимое указателя <code>u</code> и обнуляет его
<code>u. reset() u. reset(q) u. reset(nullptr)</code>	Удаляет объект, на который указывает указатель <code>u</code> . Если предоставляется встроенный указатель <code>q</code> , то <code>u</code> будет указывать на его объект. В противном случае указатель <code>u</code> обнуляется

Поскольку указатель `unique_ptr` владеет объектом, на который указывает, он не поддерживает обычного копирования и присвоения:

```
unique_ptr<string> p1( new string("Stegosaurus") );
unique_ptr<string> p2( p1 ); // ошибка: невозможно
копирование unique_ptr
unique_ptr<string> p3;
p3 = p2; // ошибка: невозможно присвоение
unique_ptr
```

Хотя указатель `unique_ptr` нельзя ни присвоить, ни скопировать, можно передать собственность от одного (неконстантного) указателя `unique_ptr` другому, вызвав функцию `release()` или `reset()`:

```
// передает собственность от p1 (указывающего на
// строку "Stegosaurus") к p2
unique_ptr<string> p2( p1.release() ); // release()
обнуляет p1
unique_ptr<string> p3( new string("Trex") );
// передает собственность от p3 к p2
p2.reset( p3.release() ); // reset() освобождает
память, на которую
// указывал указатель p2
```

Функция-член `release()` возвращает указатель, хранимый в настоящее время в указателе `unique_ptr`, и обнуляет указатель `unique_ptr`. Таким образом, указатель `p2` инициализируется указателем, хранимым в указателе `p1`, а сам указатель `p1` становится нулевым.

Функция-член `reset()` получает необязательный указатель и переустанавливает указатель `unique_ptr` на заданный указатель. Если указатель `unique_ptr` не нулевой, то объект, на который он указывает, удаляется. Поэтому вызов функции `reset()` указателя `p2` освобождает память, используемую строкой со значением `"Stegosaurus"`, передает содержимое указателя `p3` указателю `p2` и обнуляет указатель `p3`.

Вызов функции `release()` нарушает связь между указателем `unique_ptr` и объектом, который он контролирует. Зачастую указатель, возвращенный функцией `release()`, используется для инициализации или присвоения другому интеллектуальному указателю. В этом случае ответственность за управление памятью просто передается от одного интеллектуального указателя другому. Но если другой интеллектуальный указатель не используется для хранения указателя, возвращенного функцией `release()`, то ответственность за освобождения этого ресурса берет на себя программа:

```
p2.release(); // ОШИБКА: p2 не освободит память, и
```

```
указатель
                // будет потерян
    auto p = p2.release(); // ok, но следует не забыть
delete( p)
```

Передача и возвращение указателя unique_ptr

Из правила, запрещающего копирование указателя `unique_ptr`, есть одно исключение: можно копировать и присваивать те указатели `unique_ptr`, которые предстоит удалить. Наиболее распространенный пример — возвращение указателя `unique_ptr` из функции:

```
unique_ptr<int> clone( int p ) {
    // ok: явное создание unique_ptr<int> для int*
    return unique_ptr<int>( new int( p ) );
}
```

В качестве альтернативы можно также возвратить копию локального объекта:

```
unique_ptr<int> clone( int p ) {
    unique_ptr<int> ret( new int( p ) );
    // ...
    return ret;
}
```

В обоих случаях компилятор знает, что возвращаемый объект будет сейчас удален. В таких случаях компилятор осуществляет специальный вид "копирования", обсуждаемый в разделе 13.6.2.

Совместимость с прежней версией: класс auto_ptr

Прежние версии библиотеки включали класс `auto_ptr`, обладавший некоторыми, но не всеми, свойствами указателя `unique_ptr`. В частности, невозможно было хранить указатели `auto_ptr` в контейнере и возвращать их из функции.

Хотя указатель `auto_ptr` все еще присутствует в стандартной библиотеке, вместо него следует использовать указатель `unique_ptr`.

Передача функции удаления указателю unique_ptr

Подобно указателю `shared_ptr`, для освобождения объекта, на который указывает указатель `unique_ptr`, по умолчанию используется оператор `delete`. Подобно указателю `shared_ptr`, функцию удаления указателя `unique_ptr` (см. раздел 12.1.4) можно переопределить. Но по

причинам, описанным в разделе 16.1.6, способ применения функции удаления указателем `unique_ptr` отличается от такового у `shared_ptr`.

Переопределение функции удаления указателя `unique_ptr` влияет на тип и способ создания (или переустановки) объектов этого типа. Подобно переопределению оператора сравнения ассоциативного контейнера (см. раздел 11.2.2), тип функции удаления можно предоставить в угловых скобках наряду с типом, на который может указывать указатель `unique_ptr`. При создании или переустановке объекта этого типа предоставляется вызываемый объект определенного типа:

```
// p указывает на объект типа objT и использует  
объект типа delt
```

```
// для его освобождения  
// он вызовет объект по имени fcn типа delt  
unique_ptr<objT, delt> p( new objT, fcn);
```

В качестве несколько более конкретного примера перепишем программу соединения так, чтобы использовать указатель `unique_ptr` вместо указателя `shared_ptr` следующим образом:

```
void f( destination &d /* другие необходимые  
параметры */ ) {  
    connection c = connect( &d ); // открыть соединение  
    // когда p будет удален, соединение будет закрыто  
    unique_ptr<connection, decltype( end_connection )*>  
    p( &c, end_connection );  
    // использовать соединение  
    // по завершении f(), даже при исключении,  
соединение будет  
    // закрыто правильно  
}
```

Для определения типа указателя на функцию используется ключевое слово `decltype` (см. раздел 2.5.3). Поскольку выражение `decltype(end_connection)` возвращает тип функции, следует добавить символ `*`, указывающий, что используется указатель на этот тип (см. раздел 6.7).

Упражнения раздела 12.1.5

Упражнение 12.16. Компиляторы не всегда предоставляют понятные сообщения об ошибках, если осуществляется попытка скопировать или

присвоить указатель `unique_ptr`. Напишите программу, которая содержит эти ошибки, и посмотрите, как компилятор диагностирует их.

Упражнение 12.17. Какие из следующих объявлений указателей `unique_ptr` недопустимы или вероятнее всего приведут к ошибке впоследствии? Объясните проблему каждого из них.

```
int ix = 1024, *pi = &ix, *pi2 = new int(2048);  
typedef unique_ptr<int> IntP;  
( a) IntP p0(ix);           ( b) IntP p1(pi);  
( c) IntP p2(pi2);         ( d) IntP p3(&ix);  
( e) IntP p4(new int(2048)); ( f) IntP p5(p2.get());
```

Упражнение 12.18. Почему класс указателя `shared_ptr` не имеет функции-члена `release()`?



12.1.6. Класс `weak_ptr`

C++
11

Класс `weak_ptr` (табл. 12.5) представляет интеллектуальный указатель, который не контролирует продолжительность существования объекта, на который он указывает. Он только указывает на объект, который контролирует указатель `shared_ptr`. Привязка указателя `weak_ptr` к указателю `shared_ptr` не изменяет счетчик ссылок этого указателя `shared_ptr`. Как только последний указатель `shared_ptr` на этот объект будет удален, удаляется и сам объект. Этот объект будет удален, даже если останется указатель `weak_ptr` на него. Имя `weak_ptr` отражает концепцию "слабого" совместного использования объекта.

Создаваемый указатель `weak_ptr` инициализируется из указателя `shared_ptr`:

```
auto p = make_shared<int>( 42 );
weak_ptr<int> wp( p ); // wp слабо связан с p;
счетчик ссылок p неизменен
```

Здесь указатели `wp` и `p` указывают на тот же объект. Поскольку совместное использование слабо, создание указателя `wp` не изменяет счетчик ссылок указателя `p`; это делает возможным удаление объекта, на который указывает указатель `wp`.

Таблица 12.5. Функции указателя `weak_ptr`

<code>weak_ptr<T> w</code>	Обнуляет указатель <code>weak_ptr</code> , способный указывать на объект типа <code>T</code>
<code>weak_ptr<T> w(sp)</code>	Указатель <code>weak_ptr</code> на тот же объект, что и указатель <code>sp</code> типа <code>shared_ptr</code> . Тип <code>T</code> должен быть приводим к типу, на который указывает <code>sp</code>
<code>w = p</code>	Указатель <code>p</code> может иметь тип <code>shared_ptr</code> или <code>weak_ptr</code> . После присвоения <code>w</code> разделяет собственность с указателем <code>p</code>
<code>w.reset()</code>	Обнуляет указатель <code>w</code>
<code>w.use_count()</code>	Возвращает количество указателей <code>shared_ptr</code> , разделяющих собственность с указателем <code>w</code>

w. expired()	Возвращает значение <code>true</code> , когда функция <code>w. use_count()</code> должна возвратить нуль, и значение <code>false</code> в противном случае
w. lock()	Возвращает нулевой указатель <code>shared_ptr</code> , если функция <code>expired()</code> должна возвратить значение <code>true</code> ; в противном случае возвращает указатель <code>shared_ptr</code> на объект, на который указывает указатель <code>w</code>

Поскольку объект может больше не существовать, нельзя использовать указатель `weak_ptr` для непосредственного доступа к его объекту. Для этого следует вызвать функцию `lock()`. Она проверяет существование объекта, на который указывает указатель `weak_ptr`. Если это так, то функция `lock()` возвращает указатель `shared_ptr` на совместно используемый объект. Такой указатель гарантирует существование объекта, на который он указывает, по крайней мере, пока существует этот указатель `shared_ptr`. Рассмотрим пример:

```
if (shared_ptr<int> np = wp.lock()) { // true, если
    пр не нулевой
    // в if, пр совместно использует свой объект с p
}
```

Внутренняя часть оператора `if` доступна только в случае истинности вызова функции `lock()`. В операторе `if` использование указателя `pr` для доступа к объекту вполне безопасно.

Проверяемый класс указателя

Для того чтобы проиллюстрировать, насколько полезен указатель `weak_ptr`, определим вспомогательный класс указателя для нашего класса `StrBlob`. Класс указателя, назовем его `StrBlobPtr`, будет хранить указатель `weak_ptr` на переменную-член `data` класса `StrBlob`, которым он был инициализирован. Использование указателя `weak_ptr` не влияет на продолжительность существования вектора, на который указывает данный объект класса `StrBlob`. Но можно воспрепятствовать попытке доступа к вектору, которого больше не существует.

Класс `StrBlobPtr` будет иметь две переменные-члена: указатель `wptr`, который может быть либо нулевым, либо указателем на вектор в объекте класса `StrBlob`; и переменную `curr`, хранящую индекс элемента, который в настоящее время обозначает этот объект. Подобно вспомогательному классу класса `StrBlob`, у класса указателя есть функция-член `check()`, проверяющая безопасность обращения к значению `StrBlobPtr`:

```

// StrBlobPtr передает исключение при попытке
доступа к
// несуществующему элементу
class StrBlobPtr {
public:
    StrBlobPtr() : curr(0) { }
    StrBlobPtr( StrBlob &a, size_t sz = 0):
        wptr( a.data), curr(sz) { }
        std::string& deref() const;
        StrBlobPtr& incr(); // префиксная версия
private:
    // check() возвращает shared_ptr на вектор, если
проверка успешна
    std::shared_ptr<std::vector<std::string>>
        check( std::size_t, const std::string&) const;
        // хранит weak_ptr, означая возможность удаления
основного вектора
        std::weak_ptr<std::vector<std::string>> wptr;
        std::size_t curr; // текущая позиция в пределах
массива
    };

```

Стандартный конструктор создает нулевой указатель `StrBlobPtr`. Список инициализации его конструктора (см. раздел 7.1.4) явно инициализирует переменную-член `curr` нулем и неявно инициализирует указатель-член `wptr` как нулевой указатель `weak_ptr`. Второй конструктор получает ссылку на `StrBlob` и (необязательно) значение индекса. Этот конструктор инициализирует `wptr` как указатель на вектор данного объекта класса `StrBlob` и инициализирует переменную `curr` значением `sz`. Используем аргумент по умолчанию (см. раздел 6.5.1) для инициализации переменной `curr`, чтобы обозначить первый элемент. Как будет продемонстрировано, ниже параметр `sz` будет использован функцией-членом `end()` класса `StrBlob`.

Следует заметить, что нельзя связать указатель `StrBlobPtr` с константным объектом класса `StrBlob`. Это ограничение следует из того факта, что конструктор получает ссылку на неконстантный объект типа `StrBlob`.

Функция-член `check()` класса `StrBlobPtr` отличается от таковой у класса `StrBlob`, поскольку она должна проверять, существует ли еще

вектор, на который он указывает:

```
std::shared_ptr<std::vector<std::string>>
StrBlobPtr::check( std::size_t i, const std::string
&msg) const {
    auto ret = wptr.lock(); // существует ли еще
    вектор?
    if (!ret)
        throw std::runtime_error("unbound StrBlobPtr");
    if (i >= ret->size())
        throw std::out_of_range(msg);
    return ret; // в противном случае, возвратить
    shared_ptr на вектор
}
```

Так как указатель `weak_ptr` не влияет на счетчик ссылок соответствующего указателя `shared_ptr`, вектор, на который указывает `StrBlobPtr`, может быть удален. Если вектора нет, функция `lock()` возвратит нулевой указатель. В таком случае любое обращение к вектору потерпит неудачу и приведет к передаче исключения. В противном случае функция `check()` проверит переданный индекс. Если значение допустимо, функция `check()` возвратит указатель `shared_ptr`, полученный из функции `lock()`.

Операции с указателями

Определение собственных операторов рассматривается в главе 14, а пока определим функции `deref()` и `incr()` для обращения к значению и инкремента указателя класса `StrBlobPtr` соответственно.

Функция-член `deref()` вызывает функцию `check()` для проверки безопасности использования вектора и принадлежности индекса `curr` его диапазону:

```
std::string& StrBlobPtr::deref() const {
    auto p = check(curr, "dereference past end");
    return (*p)[curr]; // (*p) - вектор, на который
    указывает этот объект
}
```

Если проверка прошла успешно, то `p` будет указателем типа `shared_ptr` на вектор, на который указывает данный указатель `StrBlobPtr`. Выражение `(*p)[curr]` обращается к значению данного указателя `shared_ptr`, чтобы получить вектор, и использует оператор

индексирования для доступа и возвращения элемента по индексу curr.

Функция-член incr() также вызывает функцию check() :

```
// префикс: возвратить ссылку на объект после
инкремента
StrBlobPtr& StrBlobPtr::incr() {
    // если curr уже указывает на элемент после конца
    // контейнера,
    // его инкремент не нужен
    check(curr, "increment past end of StrBlobPtr");
    ++curr; // инкремент текущего состояния
    return *this;
}
```

Безусловно, чтобы получить доступ к переменной-члену data, наш класс указателя должен быть дружественным классу StrBlob (см. раздел 7.3.4). Снабдим также класс StrBlob функциями begin() и end(), возвращающими указатель StrBlobPtr на себя:

```
// предварительное объявление необходимо для
объявления дружественным
// классу StrBlob
class StrBlobPtr;
class StrBlob {
    friend class StrBlobPtr;
    // другие члены, как в разделе 12.1.1
    // возвратить указатель StrBlobPtr на первый и
    // следующий
    // после последнего элементы
    StrBlobPtr begin() { return StrBlobPtr(*this); }
    StrBlobPtr end()
    { auto ret = StrBlobPtr(*this, data->size());
        return ret; }
};
```

Упражнения раздела 12.1.6

Упражнение 12.19. Определите собственную версию класса StrBlobPtr и модифицируйте класс StrBlob соответствующим объявлением дружественным, а также функциями-членами begin() и end().

Упражнение 12.20. Напишите программу, которая построчно читает

исходный файл в операционной системе класса `StrBlob` и использует указатель `StrBlobPtr` для вывода каждого его элемента.

Упражнение 12.21. Функцию-член `deref()` класса `StrBlobPtr` можно написать следующим образом:

```
std::string& deref() const
{ return (*check(curr, "dereference past end"))[curr]; }
```

Какая версия по-вашему лучше и почему?

Упражнение 12.22. Какие изменения следует внести в класс `StrBlobPtr`, чтобы получить класс, применимый с типом `const StrBlob`? Определите класс по имени `ConstStrBlobPtr`, способный указывать на `const StrBlob`.



12.2. Динамические массивы

Операторы `new` и `delete` резервируют объекты по одному. Некоторым приложениям нужен способ резервировать хранилище для многих объектов сразу. Например, векторы и строки хранят свои элементы в непрерывной памяти и должны резервировать несколько элементов сразу всякий раз, когда контейнеру нужно повторное резервирование (см. раздел 9.4).

Для этого язык и библиотека предоставляют два способа резервирования всего массива объектов. Язык определяет второй вид оператора `new`, резервирующего и инициализирующего массив объектов. Библиотека предоставляет шаблон класса `allocator`, позволяющий отделять резервирование от инициализации. По причинам, описанным в разделе 12.2.2, применение класса `allocator` обычно обеспечивает лучшую производительность и более гибкое управление памятью.

У многих (возможно, у большинства) приложений нет никакой непосредственной необходимости в динамических массивах. Когда приложение нуждается в переменном количестве объектов, практически всегда проще, быстрей и безопасней использовать вектор (или другой библиотечный контейнер), как было сделано в классе `StrBlob`. По причинам, описанным в разделе 13.6, преимущества использования библиотечного контейнера даже более явны по новому стандарту. Библиотеки, поддерживающие новый стандарт, работают существенно быстрее, чем предыдущие версии.

Рекомендуем

Большинство приложений должно использовать библиотечные контейнеры, а не динамически созданные массивы. Использовать контейнер проще, так как меньше вероятность допустить ошибку управления памятью, и, вероятно, он обеспечивает лучшую производительность.

Как уже упоминалось, использующие контейнеры классы могут использовать заданные по умолчанию версии операторов копирования, присвоения и удаления (см. раздел 7.1.5). Классы, резервирующие

динамические массивы, должны определить собственные версии этих операторов для управления памятью при копировании, присвоении и удалении объектов.



Не резервируйте динамические массивы в классах, пока не прочитаете главу 13.



12.2.1. Оператор `new` и массивы

Чтобы запросить оператор `new` зарезервировать массив объектов, после имени типа следует указать в квадратных скобках количество резервируемых объектов. В данном случае оператор `new` резервирует требуемое количество объектов и (при успешном резервировании) возвращает указатель на первый из них:

```
// вызов get_size() определит количество
резервируемых целых чисел
int *ria = new int[get_size()]; // ria указывает на
первое из них
```

Значение в скобках должно иметь целочисленный тип, но не обязано быть константой.

Для представления типа массива при резервировании можно также использовать псевдоним типа (см. раздел 2.5.1). В данном случае скобки не нужны:

```
typedef int arrT[42]; // arrT - имя типа массива из
42 целых чисел
int *p = new arrT;      // резервирует массив из 42
целых чисел;
                        // p указывает на первый его
элемент
```

Здесь оператор `new` резервирует массив целых чисел и возвращает указатель на его первый элемент. Даже при том, что никаких скобок в коде нет, компилятор выполняет это выражение, используя оператор `new[]`. Таким образом, компилятор выполняет это выражение, как будто код был написан так:

```
int *p = new int[42];
```

***Резервирование массива возвращает указатель на тип
элемента***

Хотя обычно память, зарезервированную оператором `new T[]`, называют "динамическим массивом", это несколько вводит в заблуждение. Когда мы используем оператор `new` для резервирования массива, объект типа массива получен не будет. Вместо этого будет получен указатель на тип элемента массива. Даже если для определения типа массива

использовать псевдоним типа, оператор new не резервирует объект типа массива. И в данном случае резервируется массив, хотя часть [число] не видима. Даже в этом случае оператор new возвращает указатель на тип элемента.



Поскольку зарезервированная память не имеет типа массива, для динамического массива нельзя вызвать функцию begin() или end() (см. раздел 3.5.3). Для возвращения указателей на первый и следующий после последнего элементы эти функции используют размерность массива (являющуюся частью типа массива). По тем же причинам для обработки элементов так называемого динамического массива нельзя также использовать серийный оператор for.



ВНИМАНИЕ

Важно помнить, что у так называемого динамического массива нет типа массива.

Инициализация массива динамически созданных объектов

Зарезервированные оператором new объекты (будь то одиночные или их массивы) инициализируются по умолчанию. Для инициализации элементов массива по умолчанию (см. раздел 3.3.1) за размером следует расположить пару круглых скобок:

```
int *pia = new int[10];           // блок из десяти
неинициализированных
                                         // целых чисел
int *pia2 = new int[10](); // блок из десяти целых
чисел,
                                         // инициализированных по
умолчанию
                                         // значением 0
string *psa = new string[10]; // блок из десяти
пустых строк
string *psa2 = new string[10](); // блок из десяти
пустых строк
```



По новому стандарту можно также предоставить в скобках список инициализаторов элементов:

```
// блок из десяти целых чисел, инициализированных
соответствующим
// инициализатором
int *pia3 = new int[ 10 ]{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
// блок из десяти строк; первые четыре
инициализируются заданными
// инициализаторами, остальные элементы
инициализируются значением
// по умолчанию
string *psa3 = new string[ 10 ]{ "a", "an", "the",
string( 3, 'x' ) };
```

При списочной инициализации объекта типа встроенного массива (см. раздел 3.5.1) инициализаторы используются для инициализации первых элементов массива. Если инициализаторов меньше, чем элементов, остальные инициализируются значением по умолчанию. Если инициализаторов больше, чем элементов, оператор new потерпит неудачу, не зарезервировав ничего. В данном случае оператор new передает исключение типа `bad_array_new_length`. Подобно исключению `bad_alloc`, этот тип определен в заголовке `new`.



Хотя для инициализации элементов массива по умолчанию можно использовать пустые круглые скобки, в них нельзя предоставить инициализаторы для элементов. Благодаря этому факту при резервировании массива нельзя использовать ключевое слово `auto` (см. раздел 12.1.2).

Динамическое резервирование пустого массива вполне допустимо

Для определения количества резервируемых объектов можно использовать произвольное выражение:

```
size_t n = get_size(); // get_size() возвращает
количество необходимых
// элементов
int* p = new int[ n ]; // резервирует массив для
содержания элементов
```

```
for (int* q = p; q != p + n; ++q)
    /* обработка массива */ ;
```

Возникает интересный вопрос: что будет, если функция `get_size()` возвратит значение 0? Этот код сработает прекрасно. Вызов функции `new[n]` при `n` равном 0 вполне допустим, даже при том, что нельзя создать переменную типа массива размером 0:

```
char arr[0]; // ошибка: нельзя определить массив нулевой длины
```

```
char *cp = new char[0]; // ok: но обращение к значению cp невозможно
```

При использовании оператора `new` для резервирования массива нулевого размера он возвращает допустимый, а не нулевой указатель. Этот указатель гарантированно будет отличен от любого другого указателя, возвращенного оператором `new`. Он будет подобен указателю на элемент после конца (см. раздел 3.5.3) для нулевого элемента массива. Этот указатель можно использовать теми способами, которыми используется итератор после конца. Его можно сравнивать в цикле, как выше. К нему можно добавить нуль (или вычесть нуль), такой указатель можно вычесть из себя, получив в результате нуль. К значению такого указателя нельзя обратиться, в конце концов, он не указывает на элемент.

В гипотетическом цикле, если функция `get_size()` возвращает 0, то `n` также равно 0. Вызов оператора `new` зарезервирует нуль объектов. Условие оператора `for` будет ложно (`p` равно `q + n`, поскольку `n` равно 0). Таким образом, тело цикла не выполняется.

Освобождение динамических массивов

Для освобождения динамического массива используется специальная форма оператора `delete`, имеющая пустую пару квадратных скобок:

```
delete p; // p должен указывать на динамически созданный объект или
```

```
// быть нулевым
```

```
delete [] pa; // pa должен указывать на динамически созданный
```

```
// объект или быть нулевым
```

Второй оператор удаляет элементы массива, на который указывает `pa`, и освобождает соответствующую память. Элементы массива удаляются в обратном порядке. Таким образом, последний элемент удаляется первым, затем предпоследний и т.д.

При применении оператора `delete` к указателю на массив пустая пара квадратных скобок необходима: она указывает компилятору, что указатель содержит адрес первого элемента массива объектов. Если пропустить скобки оператора `delete` для указателя на массив (или предоставить их, передав оператору `delete` указатель на объект), то его поведение будет непредсказуемо.

Напомним, что при использовании псевдонима типа, определяющего тип массива, можно зарезервировать массив без использования `[]` в операторе `new`. Но даже в этом случае нужно использовать скобки при удалении указателя на этот массив:

```
typedef int arrT[ 42 ]; // arrT имя типа массив из 42
целых чисел
int *p = new arrT; // резервирует массив из 42
целых чисел; p указывает
// на первый элемент
delete [] p; // скобки необходимы,
поскольку был
// зарезервирован массив
```

Несмотря на внешний вид, указатель `p` указывает на первый элемент массива объектов, а не на отдельный объект типа `arrT`. Таким образом, при удалении указателя `p` следует использовать `[]`.



Компилятор вряд ли предупредит нас, если забыть скобки при удалении указателя на массив или использовать их при удалении указателя на объект. Программа будет выполняться с ошибкой, не предупреждая о ее причине.

Интеллектуальные указатели и динамические массивы

Библиотека предоставляет версию указателя `unique_ptr`, способную контролировать массивы, зарезервированные оператором `new`. Чтобы использовать указатель `unique_ptr` для управления динамическим массивом, после типа объекта следует расположить пару пустых скобок:

```
// up указывает на массив из десяти
неинициализированных целых чисел
unique_ptr<int[ ]> up( new int[ 10 ] );
```

```

    up.release(); // автоматически использует оператор
delete[] для
                // удаления указателя

```

Скобки в спецификаторе типа (`<int[]>`) указывают, что указатель `up` указывает не на тип `int`, а на массив целых чисел. Поскольку указатель `up` указывает на массив, при удалении его указателя автоматически используется оператор `delete[]`.

Указатели `unique_ptr` на массивы предоставляют несколько иные функции, чем те, которые использовались в разделе 12.1.5. Эти функции описаны в табл. 12.6. Когда указатель `unique_ptr` указывает на массив, нельзя использовать точечный и стрелочный операторы доступа к элементам. В конце концов, указатель `unique_ptr` указывает на массив, а не на объект, поэтому эти операторы были бы бессмысленны. С другой стороны, когда указатель `unique_ptr` указывает на массив, для доступа к его элементам можно использовать оператор индексирования:

```

for (size_t i = 0; i != 10; ++i)
    up[i] = i; // присвоить новое значение каждому из
элементов

```

Таблица 12.6. Функции указателя `unique_ptr` на массив

Операторы доступа к элементам (точка и стрелка) не поддерживаются указателями <code>unique_ptr</code> на массивы. Другие его функции неизменны	
<code>unique_ptr<T[]></code> <code>u</code>	<code>u</code> может указывать на динамически созданный массив типа <code>T</code>
<code>unique_ptr<T[]></code> <code>u(p)</code>	<code>u</code> указывает на динамически созданный массив, на который указывает встроенный указатель <code>p</code> . Тип указателя <code>p</code> должен допускать приведение к типу <code>T</code> (см. раздел 4.11.2). Выражение <code>u[i]</code> возвратит объект в позиции <code>i</code> массива, которым владеет указатель <code>u</code> . <code>u</code> должен быть указателем на массив

В отличие от указателя `unique_ptr`, указатель `shared_ptr` не оказывает прямой поддержки управлению динамическим массивом. Если необходимо использовать указатель `shared_ptr` для управления динамическим массивом, следует предоставить собственную функцию удаления:

```

// чтобы использовать указатель shared_ptr, нужно
предоставить
// функцию удаления
shared_ptr<int> sp( new int[10], [ ](int *p) {

```

```
delete[ ] p; });
sp.reset(); // использует предоставленное лямбда-
выражение, которое в
// свою очередь использует оператор delete[ ] для
освобождения массива
```

Здесь лямбда-выражение (см. раздел 10.3.2), использующее оператор `delete[]`, передается как функция удаления.

Если не предоставить функции удаления, результат выполнения этого кода непредсказуем. По умолчанию указатель `shared_ptr` использует оператор `delete` для удаления объекта, на который он указывает. Если объект является динамическим массивом, то при использовании оператора `delete` возникнут те же проблемы, что и при пропуске `[]`, когда удаляется указатель на динамический массив (см. раздел 12.2.1).

Поскольку указатель `shared_ptr` не поддерживает прямой доступ к массиву, для обращения к его элементам применяется следующий код:

```
// shared_ptr не имеет оператора индексирования и
не поддерживает
// арифметических действий с указателями
for (size_t i = 0; i != 10; ++i)
*(sp.get() + i) = i; // для доступа к встроенному
указателю
// используется функция get()
```

Указатель `shared_ptr` не имеет оператора индексирования, а типы интеллектуальных указателей не поддерживают арифметических действий с указателями. В результате для доступа к элементам массива следует использовать функцию `get()`, возвращающую встроенный указатель, который можно затем использовать обычным способом.

Упражнения раздела 12.2.1

Упражнение 12.23. Напишите программу, конкатенирующую два строковых литерала и помещающую результат в динамически созданный массив символов. Напишите программу, конкатенирующую две строки библиотечного типа `string`, имеющих те же значения, что и строковые литералы, используемые в первой программе.

Упражнение 12.24. Напишите программу, которая читает строку со стандартного устройства ввода в динамически созданный символьный массив. Объясните, как программа обеспечивает ввод данных переменного размера. Проверьте свою программу, введя строку, размер которой

превышает длину зарезервированного массива.

Упражнение 12.25. С учетом следующего оператора new, как будет удаляться указатель pa?

```
int *pa = new int[10];
```

12.2.2. Класс allocator

Важный аспект, ограничивающий гибкость оператора `new`, заключается в том, что он объединяет резервирование памяти с созданием объекта (объектов) в этой памяти. Точно так же оператор `delete` объединяет удаление объекта с освобождением занимаемой им памяти. Обычно объединение инициализации с резервированием — это именно то, что и нужно при резервировании одиночного объекта. В этом случае почти наверняка известно значение, которое должен иметь объект.

Когда резервируется блок памяти, обычно в нем планируется создавать объекты по мере необходимости. В таком случае желательно было бы отделить резервирование памяти от создания объектов. Это позволит резервировать память в больших объемах, а дополнительные затраты на создание объектов нести только тогда, когда это фактически необходимо.

Зачастую объединение резервирования и создания оказывается расточительным. Например:

```
string *const p = new string[n]; // создает n
пустых строк
string s;
string *q = p; // q указывает на первую строку
while (cin >> s && q != p + n)
    *q++ = s; // присваивает новое значение *q
const size_t size = q - p; // запомнить количество
прочитанных строк
// использовать массив
delete[] p; // p указывает на массив; не забыть
использовать delete[]
```

Этот оператор `new` резервирует и инициализирует `n` строк. Но `n` строк может не понадобиться, — вполне может хватить меньшего количества строк. В результате, возможно, были созданы объекты, которые никогда не будут использованы. Кроме того, тем из объектов, которые действительно используются, новые значения присваиваются немедленно, поверх только что инициализированных строк. Используемые элементы записываются дважды: сначала, когда им присваивается значение по умолчанию, а затем, когда им присваивается значение.

Еще важней то, что классы без стандартных конструкторов не могут быть динамически созданы как массив.

Класс allocator и специальные алгоритмы

Библиотечный класс `allocator`, определенный в заголовке `memory`, позволяет отделить резервирование от создания. Он обеспечивает не типизированное резервирование свободной области память. Операции, поддерживаемые классом `allocator`, приведены в табл. 12.7. Операции с классом `allocator` описаны в этом разделе, а типичный пример его использования — в разделе 13.5.

Подобно типу `vector`, тип `allocator` является шаблоном (см. раздел 3.3). Чтобы определить экземпляр класса `allocator`, следует указать тип объектов, которые он сможет резервировать. Когда объект `allocator` резервирует память, он обеспечивает непрерывное хранилище соответствующего размера для содержания объектов заданного типа:

```
allocator<string> alloc; // объект, способный  
резервировать строки  
auto const p = alloc.allocate( n ); // резервирует n  
незаполненных строк
```

Этот вызов функции `allocate()` резервирует память для `n` строк.

Таблица 12.7. Стандартный класс allocator и специальные алгоритмы

<code>allocator<T> a</code>	Определяет объект <code>a</code> класса <code>allocator</code> , способный резервировать память для объектов типа <code>T</code>
<code>a.allocate(n)</code>	Резервирует пустую область памяти для содержания <code>n</code> объектов типа <code>T</code>
<code>a.deallocate(p, n)</code>	Освобождает область памяти, содержащую <code>n</code> объектов типа <code>T</code> , начиная с адреса в указателе <code>p</code> типа <code>T*</code> . Указатель <code>p</code> должен быть ранее возвращен функцией <code>allocate()</code> , а размер <code>n</code> — соответствовать запрошенному при создании указателя <code>p</code> . Функцию <code>destroy()</code> следует выполнить для всех объектов, созданных в этой памяти, прежде, чем вызывать функцию <code>deallocate()</code>
<code>a.construct(p, args)</code>	Указатель <code>p</code> на тип <code>T</code> должен указывать на незаполненную область памяти; аргументы <code>args</code> передаются конструктору типа <code>T</code> , используемому для создания объекта в памяти, на которую указывает указатель <code>p</code>
<code>a.destroy(p)</code>	Выполняет деструктор (см. раздел 12.1.1) для объекта, на который указывает указатель <code>p</code> типа <code>T*</code>

Класс allocator резервирует незаполненную память

Память, которую резервирует объект класса `allocator`, не заполнена. Эта область памяти используется при создании объектов. В новой библиотеке функция-член `construct()` получает указатель и любое количество дополнительных аргументов; она создает объекты в заданной области памяти. Для инициализации создаваемого объекта используются дополнительные аргументы. Подобно аргументам функции `make_shared()` (см. раздел 12.1.1), эти дополнительные аргументы должны быть допустимыми инициализаторами объекта создаваемого типа. В частности, если типом объекта является класс, эти аргументы должны соответствовать конструктору этого класса:

```
auto q = p; // q указывает на следующий элемент
после последнего
                    // созданного
alloc.construct( q++ ); // *q - пустая
строка
alloc.construct( q++, 10, 'c' ); // *q - cccccccccc
alloc.construct( q++, "hi" ); // *q - hi!
```

В прежних версиях библиотеки функция `construct()` получала только два аргумента: указатель для создаваемого объекта и значение его типа. В результате можно было только скопировать объект в незаполненную область, но никакой другой конструктор этого типа использовать было нельзя.

Использование незаполненной области памяти, в которой еще не был создан объект, является ошибкой:

```
cout << *p << endl; // ok: использует оператор
вывода класса string
cout << *q << endl; // ошибка: q указывает на
незаполненную память!
```



ВНИМАНИЕ

Чтобы использовать память, возвращенную функцией `allocate()`, в ней следует создать объекты. Результат использования незаполненной памяти другими способами непредсказуем.

По завершении использования объектов следует удалить ранее

созданные элементы. Для этого следует вызвать функцию `destroy()` каждого созданного элемента. Функция `destroy()` получает указатель и запускает деструктор (см. раздел 12.1.1) указанного объекта:

```
while ( q != p )
    alloc.destroy( --q ); // освободить фактически
зарезервированные строки
```

В начале цикла `q` указывает на следующий элемент после последнего заполненного. Перед вызовом функции `destroy()` осуществляется декремент указателя `q`. Таким образом, при первом вызове функции `destroy()` указатель `q` указывает на последний созданный элемент. Первый элемент удаляется на последней итерации, после которой `q` станет равен `p` и цикл закончится.



Удалять можно только те элементы, которые были фактически созданы.

Как только элементы удалены, память можно повторно использовать для содержания других строк или возвратить их операционной системе. Для освобождения памяти используется функция `deallocate()`:

```
alloc.deallocate( p, n );
```

Указатель, передаваемый функции `deallocate()`, не может быть нулевым; он должен указывать на область памяти, зарезервированной функцией `allocate()`. Кроме того, переданный ей аргумент размера должен совпадать с размером, использованным при вызове функции `allocate()`, зарезервировавшим область памяти, на которую указывает указатель.

Алгоритмы копирования и заполнения неинициализированной памяти

В дополнение к классу `allocator` библиотека предоставляет два алгоритма, способных создавать объекты в неинициализированной памяти. Эти функции описаны в табл. 12.8 и определены в заголовке `memory`.

Таблица 12.8. Алгоритмы, связанные с классом `allocator`

Эти функции создают элементы по назначению, а не присваивают их	
	Копирует элементы из исходного диапазона, обозначенного итераторами <code>b</code> и <code>e</code> , в незаполненную

<code>uninitialized_copy(b, e, b2)</code>	память, обозначенную итератором <code>b2</code> . Память, обозначенная итератором <code>b2</code> , должна быть достаточно велика для содержания копии элементов из исходного диапазона
<code>uninitialized_copy_n(b, n, b2)</code>	Копирует <code>n</code> элементов, начиная с обозначенного итератором <code>b</code> в незаполненную память, начиная с позиции <code>b2</code>
<code>uninitialized_fill(b, e, t)</code>	Создает объекты в диапазоне незаполненной памяти, обозначенной итераторами <code>b</code> и <code>e</code> как копию <code>t</code>
<code>uninitialized_fill_n(b, n, t)</code>	Создает <code>n</code> объектов, начиная с <code>b</code> . Итератор <code>b</code> должен обозначать незаполненную память достаточного размера для содержания заданного количества объектов

Предположим, например, что имеется вектор целых чисел, который необходимо скопировать в динамическую память. Память будет резервироваться дважды для каждого целого числа в векторе. Первую половину вновь зарезервированной памяти заполним копиями элементов из исходного вектора. Элементы второй половины заполним заданным значением:

```
// зарезервировать вдвое большее элементов, чем хранения в vi
auto p = alloc.allocate( vi.size() * 2);
// создать элементы, начиная с p как копии элементов в vi
auto q = uninitialized_copy( vi.begin(), vi.end(), p);
// инициализировать остальные элементы значением 42
uninitialized_fill_n(q, vi.size(), 42);
```

Подобно алгоритму `copy()` (см. раздел 10.2.2), алгоритм `uninitialized_copy()` получает три итератора. Первые два обозначают исходную последовательность, а третий обозначает получателя, в который будут скопированы эти элементы. Итератор назначения, переданный алгоритму `uninitialized_copy()`, должен обозначить незаполненную память. В отличие от алгоритма `copy()`, алгоритм `uninitialized_copy()` создает элементы в своем получателе.

Подобно алгоритму `copy()`, алгоритм `uninitialized_copy()` возвращает (приращенный) итератор назначения. Таким образом, вызов функции `uninitialized_copy()` возвращает указатель на следующий элемент после последнего заполненного. В данном примере этот указатель

сохраняется в переменной `q`, передаваемой функции `uninitialized_fill_n()`. Эта функция, как и функция `fill_n()` (см. раздел 10.2.2), получает указатель на получателя, количество и значение. Она создает заданное количество объектов из заданного значения в позиции, начиная с заданной получателем.

Упражнения раздела 12.2.2

Упражнение 12.26. Перепишите программу из начала раздела, используя класс `allocator`.



12.3. Использование библиотеки: программа запроса текста

Для завершения обсуждения библиотеки реализуем простую программу текстового запроса. Она позволит пользователю искать в заданном файле слова, которые могли встречаться в нем. Результатом запроса будет количество экземпляров слова и список строк, в которых оно присутствует. Если слово встречается несколько раз в той же строке, то она отображается только однажды. Строки отображаются в порядке возрастания, т.е. строка номер 7 отображается перед строкой номер 9 и т.д.

Например, прочитав файл, содержащий начало этой главы и запустив поиск слова `element`, программа должна создать следующий вывод:

```
element occurs 112 times
  (line 36) A set element contains only a key;
  (line 158) operator creates a new element
  (line 160) Regardless of whether the element
  (line 168) When we fetch an element from a map, we
    (line 214) If the element is not found, find
returns
```

Далее следует примерно 100 строк, также содержащих слово `element`.



12.3.1. Проект программы

Наилучший способ начать проект программы — это составить перечень ее функциональных возможностей. Зная, какие именно функции необходимо обеспечить, значительно легче разобраться, какие именно структуры данных понадобятся. Итак, начнем с требований, которым должна удовлетворять разрабатываемая программа.

- Читая ввод, программа должна запоминать строку (строки), в которой присутствует искомое слово. Следовательно, программа должна читать ввод построчно и разделять прочитанные строки на отдельные слова
- При создании вывода программа должна:
 - получать номера строк, содержащих искомое слово;

- нумеровать строки в порядке возрастания без дубликатов;
- отображать текст исходного файла по заданному номеру строки.

Эти требования можно выполнить с помощью библиотечных средств.

- Для хранения копии всего входного файла используем вектор `vector<string>`. Каждая строка входного файла станет элементом этого вектора. При необходимости вывода строку можно будет выбрать, используя ее номер как индекс.

- Для разделения строки на слова используем строковый поток `istringstream` (см. раздел 8.3).

- Для хранения номеров строк, в которых присутствует искомое слово, используем контейнер `set`. Это гарантирует, что каждая строка будет присутствовать только однажды, а номера строки будут храниться в порядке возрастания.

- Для связи каждого слова с набором номеров строк, в которых присутствует искомое слово, используем контейнер `map`. Это позволит выбрать соответствующий набор для каждого заданного слова.

По рассматриваемым вскоре причинам в решении будет также использован указатель `shared_ptr`.

Структуры данных

Несмотря на то что программу можно было бы написать, используя контейнеры `vector`, `set` и `map` непосредственно, полезней определить более абстрактное решение. Для начала разработаем класс для содержания входного файла, чтобы упростить запрос. Этот класс, `TextQuery`, будет содержать вектор и карту. Вектор будет содержать текст входного файла, а карта ассоциировать каждое слово в этом файле с набором номеров строк, в которых присутствует искомое слово. У этого класса будет конструктор, читающий заданный входной файл, и функция, обрабатывающая запросы.

Работа функции обработки запроса довольно проста: она просматривает свою карту в поисках искомого слова. Трудней всего решить, что должна возвращать функция запроса. Если известно, что слово найдено, необходимо выяснить, сколько раз оно встретилось, в строках с какими номерами и каков текст каждой строки с этими номерами.

Проще всего вернуть все эти данные, определив второй класс, назовем его `QueryResult`, который и будет содержать результаты запроса. У этого класса будет функция `print()`, выводящая результаты, хранимые в объекте класса `QueryResult`.

Совместное использование данных классами

Класс `QueryResult` предназначен для представления результатов запроса. Эти результаты включают набор номеров строк, связанных с искомым словом, и текст соответствующих строк из входного файла. Эти данные хранятся в объектах типа `TextQuery`.

Поскольку данные, необходимые объекту класса `QueryResult`, хранятся в объекте класса `TextQuery`, необходимо решить, как получить к ним доступ. Можно было бы скопировать набор номеров строк, но это может оказаться слишком дорого. Кроме того, копировать вектор не хотелось бы потому, что это повлечет за собой копирование всего файла для вывода лишь его части (как обычно и бывает).

Избежать копирования можно, возвратив итераторы (или указатели) на содержимое объекта `TextQuery`. Но с этим подходом связана проблема: что если объект класса `TextQuery` будет удален до объекта класса `QueryResult`? В этом случае объект класса `QueryResult` ссылался бы на данные в больше не существующем объекте.

Это требует синхронизации продолжительности существования объекта класса `QueryResult` с объектом класса `TextQuery`, результата которого он представляет. Таким образом, эти два класса концептуально "совместно используют" данные. Для отражения совместного использования этих структур данных используем указатели `shared_ptr` (см. раздел 12.1.1).

Применение класса `TextQuery`

При разработке класса может быть полезно написать использующие его программы, прежде чем фактически реализовать его члены. Таким образом можно выяснить, какие функции ему необходимы. Например, следующая программа использует проектируемые классы `TextQuery` и `QueryResult`. Эта функция получает поток класса `ifstream` для подлежащего обработке файла и взаимодействует с пользователем, выводя результаты по заданному слову:

```
void runQueries(ifstream &infile) {
    // infile - поток ifstream для входного файла
    TextQuery tq(infile); // хранит файл и строит
    карту запроса
    // цикл взаимодействия с пользователем:
    приглашение ввода искомого
    // слова и вывод результатов
    while (true) {
```

```

cout << "enter word to look for, or q to quit: ";
string s;
// остановиться по достижении конца файла или при
встрече
// символа 'q' во вводе
if (!(cin >> s) || s == "q") break;
// выполнить запрос и вывести результат
print(cout, tq.query(s)) << endl;
}
}

```

Начнем с инициализации объекта `tq` класса `TextQuery` данными из переданного потока `ifstream`. Конструктор `TextQuery()` читает файл в свой вектор и создает карту, связывающую слова из ввода с номерами строк, в которых они присутствуют.

Цикл `while` (непрерывно) запрашивает у пользователя искомое слово и выводит полученные результаты. Условие цикла проверяет литерал `true` (см. раздел 2.1.3), поэтому оно всегда истинно. Для выхода из цикла используется оператор `break` (см. раздел 5.5.1) в операторе `if`. Он проверяет успешность чтения. Если оно успешно, проверяется также, не ввел ли пользователь символ "q", желая завершить работу. Получив искомое слово, запрашиваем его поиск у объекта `tq`, а затем вызываем функцию `print()` для вывода результата поиска.

Упражнения раздела 12.3.1

Упражнение 12.27. Классы `TextQuery` и `QueryResult` используют только те возможности, которые уже были описаны ранее. Не заглядывая вперед, напишите собственные версии этих классов.

Упражнение 12.28. Напишите программу, реализующую текстовые запросы, не определяя классы управления данными. Программа должна получать файл и взаимодействовать с пользователем, запрашивая слова, искомые в этом файле. Используйте контейнеры `vector`, `map` и `set` для хранения данных из файла и создания результатов запросов.

Упражнение 12.29. Перепишите цикл взаимодействия с пользователем, используя цикл `do while` (см. раздел 5.4.4). Объясните, какая версия предпочтительней и почему.



12.3.2. Определение классов программы запросов

Начнем с определения класса `TextQuery`. Пользователь создает объекты этого класса, предостав员я поток `istream` для чтения входного файла. Этот класс предоставляет также функцию `query()`, которая получает строку и возвращает объект класса `QueryResult`, представляющий строки, в которых присутствует искомое слово.

Переменные-члены класса должны учитывать совместное использование с объектами класса `QueryResult`. Класс `QueryResult` совместно использует вектор, представляющий входной файл и наборы, содержащие номера строк, связанные с каждым словом во вводе. Следовательно, у нашего класса есть две переменные-члена: указатель `shared_ptr` на динамически созданный вектор, содержащий входной файл, а также карта строк и указателей `shared_ptr<set>`. Карта ассоциирует каждое слово в файле с динамическиенным набором, содержащим номера строк, в которых присутствует это слово.

Чтобы сделать код немного понятней, определим также тип-член (см. раздел 7.3.1) для обращения к номерам строк, которые являются индексами вектора строк:

```
class QueryResult; // объявление необходимого типа
// возвращаемого
// значения функции запроса
class TextQuery {
public:
    using line_no = std::size_type;
    TextQuery( std::ifstream& );
    QueryResult query( const std::string& ) const;
private:
    std::shared_ptr<std::vector<std::string>> file; // исходный файл
    // сопоставить каждое слово с набором строк, в
    // которых присутствует
    // это слово
    std::map<std::string,
```

```
    std::shared_ptr<std::set<line_no>>> wm;
};
```

Самая трудная часть этого кода — разобраться в именах классов. Как обычно, для кода из файла заголовка применяется часть `std::`, указывающая имя библиотеки (см. раздел 3.1). Но в данном случае частое повторение имени `std::` делает код немного менее понятным для чтения. Рассмотрим пример:

```
std::map<std::string,
std::shared_ptr<std::set<line_no>>> wm;
```

Его будет проще понять, переписав так:

```
map<string, shared_ptr<set<line_no>>> wm;
```

Конструктор `TextQuery()`

Конструктор `TextQuery()` получает поток `ifstream`, позволяющий читать строки по одной:

```
// прочитать входной файл, создать карту строк и их
номеров
TextQuery::TextQuery(ifstream &is) : file(new
vector<string>) {
    string text;
    while (getline(is, text)) { // для каждой строки в
файле
        file->push_back(text); // запомнить эту
строку текста
        int n = file->size() - 1; // номер текущей
строки
        istringstream line(text); // разделить строку на
слова
        string word;
        while (line >> word) { // для каждого слова в
этой строке
            // если слова еще нет в wm, индексация добавляет
новый
            // элемент
            auto &lines = wm[word]; // lines - это
shared_ptr
            if (!lines) // этот указатель - вначале нулевой,
когда
                // встречается слово
```

```

        lines.reset( new set<line_no>); // резервирует
новый набор
    lines->insert( n); // вставить номер этой строки
}
}
}

```

Список инициализации конструктора резервирует новый вектор для содержания текста из входного файла. Функция `getline()` используется для чтения из файла по одной строке за раз и их помещения в вектор. Поскольку `file` — это указатель `shared_ptr`, используем оператор `->` для обращения к его значению, чтобы вызвать функцию `push_back()` для того элемента вектора, на который указывает указатель `file`.

Затем поток `istringstream` (см. раздел 8.3) используется для обработки каждого слова только что прочитанной строки. Внутренний цикл `while` использует оператор ввода класса `istringstream` для чтения каждого слова текущей строки в строку `word`. В цикле `while` используется оператор индексирования карты для доступа к связанному со словом указателю `shared_ptr<set>` и связи ссылки `lines` с этим указателем. Обратите внимание, что `lines` — это ссылка, поэтому внесенные в нее изменения будут сделаны с элементом карты `wm`.

Если слова еще нет в карте, оператор индексирования добавит строку `word` в карту `wm` (см. раздел 11.3.4). Ассоциируемый со строкой `word` элемент инициализирован значением по умолчанию. Это значит, что ссылка `lines` будет нулевой, если оператор индексирования добавит строку `word` в карту `wm`. Если ссылка `lines` будет нулевой, резервируем новый набор и вызываем функцию `reset()` для обновления указателя `shared_ptr`, на который ссылается ссылка `lines`, чтобы он указывал на этот только что созданный набор.

Независимо от того, был ли создан новый набор, происходит вызов функции `insert()`, добавляющей текущий номер строки. Поскольку `lines` — это ссылка, вызов функции `insert()` добавляет элемент в набор карты `wm`. Если данное слово встречается несколько раз в той же строке, вызов функции `insert()` не делает ничего.

Класс QueryResult

Класс `QueryResult` обладает тремя переменными-членами: строка, представляющая слово, указатель `shared_ptr` на вектор, содержащий входной файл; и указатель `shared_ptr` на набор номеров строк, в

которых присутствует это слово. Его единственная функция-член — конструктор, инициализирующий эти три члена:

```
class QueryResult {
    friend std::ostream& print(std::ostream&, const
QueryResult&);
public:
    QueryResult( std::string s,
                  std::shared_ptr<std::set<line_no>> p,
                  std::shared_ptr<std::vector<std::string>
f):
        sought(s), lines(p), file(f) { }
private:
    std::string sought; // слово, представляющее
запрос
    std::shared_ptr<std::set<line_no>> lines; // // номера строк
    std::shared_ptr<std::vector<std::string>> file; // // входной файл
};
```

Единственная задача конструктора — сохранить свои аргументы в соответствующих переменных-членах, что он и делает в списке инициализации конструктора (см. раздел 7.1.4).

Функция *query()*

Функция `query()` получает строку, которую она использует для поиска соответствующего набора номеров строк в карте. Если строка найдена, функция `query()` создает объект класса `QueryResult` из заданной строки, переменной-члена `file` класса `TextQuery` и набора, извлеченного из карты `wm`.

Единственный вопрос: что следует возвратить, если заданная строка не найдена? В данном случае никакого набора возвращено не будет. Решим эту проблему, определив локальный статический объект, являющийся указателем `shared_ptr` на пустой набор номеров строк. Когда слово не найдено, возвратим копию этого указателя `shared_ptr`:

```
QueryResult  
TextQuery::query( const string &sought) const {  
    // возвратить указатель на этот набор, если  
искомое слово не найдено
```

```

        static      shared_ptr<set<line_no>>      nodata( new
set<line_no> );
        // использовать find( ) но не индексировать, чтобы
избежать
        // добавления слова в карту wm!
        auto loc = wm.find( sought );
        if ( loc == wm.end( ) )
            return QueryResult( sought, nodata, file ); // не
найдено
        else
            return QueryResult( sought, loc->second, file );
    }

```

Вывод результатов

Функция print() выводит заданный объект класса QueryResult в заданный поток:

```

ostream &print( ostream &os, const QueryResult &qr)
{
    // если слово найдено, вывести количество и все
вхождения
    os << qr.sought << " occurs " << qr.lines->size()
<< " "
        << make_plural( qr.lines->size( ), "time", "s" )
<< endl;
    // вывести каждую строку, в которой присутствует
слово
    for ( auto num : *qr.lines ) // для каждого элемента
в наборе
        // не путать пользователя с номерами строк,
начинающимися с 0
        os << "\t (line " << num + 1 << ") "
            << *( qr.file->begin( ) + num ) << endl;
    return os;
}

```

Для отчета о количестве найденных соответствий используем функцию size() набора, на который ссылается qr.lines. Поскольку этот набор контролируется указателем shared_ptr, следует помнить об обращении к значению lines. Чтобы вывести слово time или times, в зависимости от того, равен ли размер 1, используем функцию make_plural() (см.

раздел 6.3.2).

Цикл `for` перебирает набор, на который ссылается `lines`. Тело цикла `for` выводит номер строки, откорректированный так, как привычно человеку. Числа в наборе являются индексами элементов в векторе, их нумерация начинается с нуля. Но большинство пользователей привыкли к тому, что первая строка имеет номер 1, поэтому будем систематически добавлять 1 к номерам строк, чтобы отображать их в общепринятой форме.

Используем номер строки для выбора строк из вектора, на который указывает указатель-член `file`. Помните, что при добавлении числа к итератору будет получен элемент на столько же элементов далее (см. раздел 3.4.2). Таким образом, часть `file->begin() + num` дает номер элемента от начала вектора, на который указывает `file`.

Обратите внимание: эта функция правильно обрабатывает случай, когда слово не найдено. В данном случае набор будет пуст. Первый оператор вывода заметит, что слово встретилось нуль раз. Поскольку `*res.lines` пуст, цикл `for` не выполнится ни разу.

Упражнения раздела 12.3.2

Упражнение 12.30. Определите собственные версии классов `TextQuery` и `QueryResult`, а также выполните функцию `runQueries()` из раздела 12.3.1.

Упражнение 12.31. Что будет, если для хранения номеров строк использовать вектор вместо набора? Какой подход лучше? Почему?

Упражнение 12.32. Перепишите классы `TextQuery` и `QueryResult` так, чтобы для хранения входного файла вместо вектора `vector<string>` использовался класс `StrBlob`.

Упражнение 12.33. В главе 15 программа запроса будет дополнена, и классу `QueryResult` понадобятся дополнительные члены. Добавьте функции-члены по имени `begin()` и `end()`, возвращающие итераторы для набора номеров строк, возвращенных данным запросом, и функцию-член `get_file()`, возвращающую указатель `shared_ptr` на файл в объекте `QueryResult`.

Резюме

В языке C++ для резервирования памяти используется оператор `new`, а для освобождения — оператор `delete`. Библиотека определяет также класс `allocator`, чтобы резервировать пустые блоки динамической

памяти.

Резервирующие динамическую память программы отвечают за ее освобождение. Освобождение динамической памяти — богатейший источник ошибок: память может быть не освобождена никогда или может быть освобождена, когда еще есть указатели на нее. Новая библиотека определяет классы интеллектуальных указателей (`shared_ptr`, `unique_ptr` и `weak_ptr`), делающие работу с динамической памятью намного более безопасной. Интеллектуальный указатель автоматически освобождает память, как только удаляется последний указатель на нее. В современных программах C++ следует использовать именно интеллектуальные указатели.

Термины

Деструктор (destructor). Специальная функция-член, которая освобождает занятую объектом память, когда он выходит из области видимости или удаляется.

Динамическая память (free store). Пул памяти, доступный программе для хранения объектов, создаваемых динамически.

Динамически созданный объект (dynamically allocated object). Объект, который создан в динамической памяти. Такие объекты существуют до тех пор, пока не будут удалены из динамической памяти явно или пока программа не завершит работу.

Интеллектуальный указатель `shared_ptr`. Интеллектуальный указатель, обеспечивающий совместную собственность: объект освобождается, когда удаляется последний указатель `shared_ptr` на тот объект.

Интеллектуальный указатель `unique_ptr`. Интеллектуальный указатель, обеспечивающий единоличную собственность: объект освобождается, когда удаляется указатель `unique_ptr` на этот объект. Указатель `unique_ptr` не может быть скопирован или присвоен непосредственно.

Интеллектуальный указатель `weak_ptr`. Интеллектуальный указатель на объект, управляемый указателем `shared_ptr`. Указатель `shared_ptr` не учитывает указатель `weak_ptr`, принимая решение об освобождении своего объекта.

Интеллектуальный указатель (smart pointer). Библиотечный тип, действует как указатель, но допускающий проверку на безопасность использования. Тип сам заботится об освобождении памяти, когда это

нужно.

Класс allocator. Библиотечный класс, резервирующий области памяти.

Оператор delete. Освобождает участок памяти, зарезервированный оператором new. Оператор delete р освобождает объект, а delete [] р — массив, на который указывает указатель р. Указатель р может быть пуст или указывать на область память, зарезервированную оператором new.

Оператор new. Резервирует область в динамической памяти. Оператор new Т резервирует область памяти, создает в ней объект типа Т и возвращает указатель на этот объект. Если Т — тип массива, оператор new возвращает указатель на его первый элемент. Аналогично оператор new [n] Т резервирует n объектов типа Т и возвращает указатель на первый элемент массива. По умолчанию вновь созданный объект инициализируется значением по умолчанию. Но можно также предоставить инициализаторы.

Потерянный указатель (dangling pointer). Указатель, содержащий адрес области памяти, в которой уже нет объекта. Потерянный указатель является весьма распространенным источником ошибок в программе, причем такие ошибки крайне трудно обнаружить.

Размещающий оператор new (placement new). Форма оператора new, получающая в круглых скобках дополнительные аргументы после ключевого слова new; например, синтаксис new (nothrow) int устанавливает, что оператор new не должен передавать исключения.

Распределяемая память (heap). То же, что и динамическая память.

Счетчик ссылок (reference count). Счетчик, отслеживающий количество пользователей, совместно использующих общий объект. Используется интеллектуальными указателями, чтобы узнать, когда безопасно освобождать память, на которую они указывают.

Функция удаления (deleter). Функция, передаваемая интеллектуальному указателю, для использования вместо оператора delete при освобождении объекта, на который указывает указатель.

Часть III

Инструменты для разработчиков классов

Классы — основная концепция языка C++. Подробное рассмотрение определения классов начато в главе 7. Она затрагивает такие фундаментальные для классов темы, как область видимости класса, сокрытие данных и конструкторы. Она познакомила также с важнейшими средствами класса: функциями-членами, неявным указателем `this`, дружественными отношениями, а также членами `const`, `static` и `mutable`. В этой части дополним тему классов, рассмотрев управление копированием, перегрузку операторов, наследование и шаблоны.

Как уже упоминалось, в классах C++ определяются конструкторы, контролирующие происходящее при инициализации объектов класса. Классы контролируют также то, что происходит при копировании, присвоении, перемещении и удалении объектов. В этом отношении язык C++ отличается от других языков, большинство из которых не позволяет разработчикам классов контролировать эти операции. В главе 13 будут рассмотрены эти темы, а также две важные концепции, введенные новым стандартом: ссылки на *r*-значения и операции перемещения.

Глава 14 посвящена перегрузке операторов, позволяющей использовать операнды типа классов со встроенными операторами. Перегрузка оператора — это один из способов, которым язык C++ позволяет создавать новые типы, столь же интуитивно понятные в использовании, как и встроенные типы.

Среди доступных для перегрузки операторов класса есть оператор вызова функции. Объекты таких классов можно "вызывать" точно так же, как если бы они были функциями. Рассмотрим также новые библиотечные средства, облегчающие использование различных типов вызываемых объектов единообразным способом.

Эта глава завершается рассмотрением еще одного специального вида функций-членов класса — операторов преобразования. Эти операторы определяют неявные преобразования из объектов типа класса. Компилятор применяет эти преобразования в тех же контекстах (и по тем же причинам), что и преобразования встроенных типов.

Последние две главы этой части посвящены поддержке языком C++ объектно-ориентированного и обобщенного программирования.

Глава 15 рассматривает наследование и динамическое связывание.

Наряду с абстракцией данных наследование и динамическое связывание — это основы объектно-ориентированного программирования. Наследование облегчает определение связанных типов, а динамическое связывание позволяет писать независимый от типов код, способный игнорировать различия между типами, которые связаны наследованием.

Глава 16 посвящена шаблонам классов и функций. Шаблоны позволяют писать обобщенные классы и функции, которые не зависят от типов. Новый стандарт ввел множество новых средств, связанных с шаблонами: шаблоны с переменным количеством аргументов, псевдонимы типов шаблона и новые способы контроля создания экземпляра.

Создание собственных объектно-ориентированных или обобщенных типов требует довольно хорошего понимания языка C++. К счастью, для их использования это не обязательно. Например, стандартная библиотека интенсивно использует средства, которые рассматриваются только в главах 15 и 16, но библиотечные типы и алгоритмы использовались уже с самого начала книги, даже без объяснения их реализации.

Поэтому читатели должны понимать, что часть III посвящена довольно сложным средствам. Написание шаблонов и объектно-ориентированных классов требует хорошего понимания основ языка C++ и глубокого знания того, как определяют базовые классы.

Глава 13

Управление копированием

Как упоминалось в главе 7, каждый класс является определением нового типа и операций, которые можно выполнять с объектами этого типа. В этой главе упоминалось также о том, что классы могут определять конструкторы, которые контролируют происходящее при создании объектов данного типа.

В этой главе мы изучим то, как классы могут контролировать происходящее при копировании, присвоении, перемещении и удалении объектов данного типа. Для этого классы имеют специальные функции-члены: *конструктор копий*, *конструктор перемещения*, *оператор присвоения копии*, *оператор присваивания при перемещении* и *деструктор*.

При определении класса разработчик (явно или неявно) определяет происходящее при копировании, перемещении, присвоении и удалении объектов данного класса. Класс контролирует эти операции, определяя пять специальных функций-членов: *конструктор копий* (*copy constructor*), *оператор присвоения копии* (*copy-assignment operator*), *конструктор перемещения* (*move constructor*), *оператор присваивания при перемещении* (*move-assignment operator*) и *деструктор* (*destructor*). Конструкторы копирования и перемещения определяют происходящее при инициализации объекта данными из другого объекта того же типа. Операторы копирования и присваивания при перемещении определяют происходящее при присвоении объекта данного класса другому объекту того же класса. Деструктор определяет происходящее в момент, когда объект данного типа прекращает существование. Все эти операции вместе мы будем называть *управлением копированием* (*copy control*).

Если класс определяет не все функции-члены управления копированием, компилятор сам определит недостающие. В результате многие классы могут не определять управление копированием (см. раздел 7.1.5). Но некоторые классы не могут полагаться на заданные по умолчанию определения. Зачастую наиболее трудная часть реализации операций управления копированием — это принятие решения об их необходимости.



Управление копированием — это важнейшая часть определения любого класса C++. У начинающих программистов C++ зачастую возникают затруднения при необходимости определения действий, происходящих при копировании, перемещении, присвоении и удалении объектов. Это затруднение обусловлено тем, что задавать их не обязательно, компилятор вполне может создать их сам, хотя результат этих действий может быть не совсем таким, как хотелось бы.

13.1. Копирование, присвоение и удаление

Начнем с наиболее простых операций: конструктора копий, оператора присвоения копии и деструктора. Операции перемещения (введенные новым стандартом) рассматриваются в разделе 13.6.



13.1.1. Конструктор копий

Если первый параметр конструктора — ссылка на тип класса, а все дополнительные параметры имеют значения по умолчанию, то это конструктор копий:

```
class Foo {  
public:  
    Foo(); // стандартный конструктор  
    Foo(const Foo&); // конструктор копий  
    // ...  
}
```

По причинам, которые будут описаны ниже, первый параметр должен иметь ссылочный тип. Он почти всегда является ссылкой на константу, хотя вполне можно определить конструктор копий, получающий ссылку на не константу. При некоторых обстоятельствах конструктор копий используется неявно. Следовательно, конструктор копий обычно не следует объявлять как `explicit` (см. раздел 7.5.4).

Синтезируемый конструктор копий

Если конструктор копий не определен для класса явно, компилятор синтезирует его сам. В отличие от синтезируемого стандартного конструктора (см. раздел 7.1.4), конструктор копий синтезируется, даже если определены другие конструкторы.

Как будет продемонстрировано в разделе 13.1.6, *синтезируемый конструктор копий* (*synthesized copy constructor*) некоторых классов препятствует копированию объектов этого типа. В противном случае синтезируемый конструктор копий осуществляет *почленное копирование* (*memberwise copy*) членов своего аргумента в создаваемый объект (см. раздел 7.1.5). Компилятор по очереди копирует каждую нестатическую переменную-член заданного объекта в создаваемый.

Способ копирования каждой переменной-члена определяет ее тип: для типов класса применяется конструктор копий этого класса, а члены встроенного типа копируются непосредственно. Хотя нельзя непосредственно скопировать массив (см. раздел 3.5.1), синтезируемый конструктор копий копирует члены типа массива поэлементно. Элементы типа класса копируются с использованием конструкторов копий элементов.

Например, синтезируемый конструктор копий для класса `Sales_data` эквивалентен следующему:

```
class Sales_data {
public:
    // другие члены и конструкторы как прежде
    // объявление, эквивалентное синтезируемому
    // конструктору копий
    Sales_data( const Sales_data& );
private:
    std::string bookNo;
    int units_sold = 0;
    double revenue = 0.0;
};

// эквивалент конструктора копий, синтезированный
// для класса Sales_data
Sales_data::Sales_data( const Sales_data &orig ):
    bookNo( orig.bookNo ), // использование конструктора
    // копий string
    units_sold( orig.units_sold ), // копирует
    orig.units_sold
    revenue( orig.revenue ) // копирует
    orig.revenue
{ } // пустое тело
```

Инициализация копией

Теперь можно полностью рассмотреть различия между прямой инициализацией и инициализацией копией (см. раздел 3.2.1):

```
string dots(10, '.'); // прямая
инициализация
string s(dots); // прямая
инициализация
strings2 = dots; // инициализация копией
```

```
string null_book = "9-999-99999-9"; //  
инициализация копией  
string nines = string(100, '9'); //  
инициализация копией
```

При прямой инициализации от компилятора требуется использовать обычный выбор функции (см. раздел 6.4) для подбора конструктора, наилучшим образом соответствующего предоставленным аргументам. Когда используется *инициализация копией* (copy initialization), от компилятора требуется скопировать правый операнд в создаваемый объект, осуществляя преобразования в случае необходимости (см. раздел 7.5.4).

Инициализация копией обычно использует конструктор копий. Но, как будет продемонстрировано в разделе 13.6.2, если у класса есть конструктор перемещения, то инициализация копией иногда использует конструктор перемещения вместо конструктора копий, а пока достаточно знать, что при инициализации копией требуется либо конструктор копий, либо конструктор перемещения.

Инициализация копией осуществляется не только при определении переменных с использованием оператора `=`, но и при:

- передаче объекта как аргумента параметру не ссылочного типа;
- возвращении объекта из функции с не ссылочным типом возвращаемого значения;
- инициализации списком в скобках элементов массива или членов агрегатного класса (см. раздел 7.5.5)

Некоторые классы используют также инициализацию копией для резервируемых объектов. Например, библиотечные контейнеры инициализируют копией свои элементы при инициализации контейнера, либо при вызове функции `insert()` или функции `push()` элемента (см. раздел 9.3.1). Элементы, созданные функцией `emplace()`, напротив, отличаются прямой инициализацией (см. раздел 9.3.1).

Параметры и возвращаемые значения

Во время вызова функции с параметрами не ссылочного типа осуществляется инициализация копией (см. раздел 6.2.1). Точно так же, когда у функции не ссылочный тип возвращаемого значения, возвращаемое значение используется в точке вызова для инициализации копией результата оператора вызова (см. раздел 6.3.2).

Тот факт, что конструктор копий используется для инициализации не ссылочных параметров типа класса, объясняет, почему собственный параметр конструктора копий должен быть ссылкой. Если бы этот

параметр не был ссылкой, то вызов не был бы успешным — при вызове конструктора копий должен быть использован конструктор копий для копирования аргумента, но для копирования аргумента следует вызвать конструктор копий и так далее до бесконечности.

Ограничения на инициализацию копией

Как уже упоминалось, используется ли инициализация копией или прямая инициализация, если используется инициализатор, то потребуется преобразование в явный конструктор (см. раздел 7.5.4):

```
vector<int> v1( 10 ); // ok: прямая инициализация
vector<int> v2 = 10; // ошибка: конструктор,
получающий размер,
                                // является явным
void f( vector<int> ); // параметр f()
инициализируется копией
f( 10 ); // ошибка: нельзя использовать явный
конструктор для
                                // копирования аргумента
f( vector<int>( 10 ) ); // ok: непосредственно создать
временный вектор
                                // из int
```

Прямая инициализация вектора `v1` корректна, но на первый взгляд эквивалентная инициализация копией вектора `v2` ошибочна, поскольку конструктор вектора, получающий один параметр размера, является явным. По тем же причинам недопустима инициализация копией вектора `v2` — нельзя неявно использовать явный конструктор при передаче аргумента или возвращении значения из функции. Если нужно использовать явный конструктор, то сделать это следует явно, как в последней строке примера, приведенного выше.

Компилятор может обойти конструктор копий

Во время инициализации копией компилятору можно (но не обязательно) пропустить конструктор копий или перемещения и создать объект непосредственно. Таким образом, код

```
string null_book = "9-999-99999-9"; // 
инициализация копией
компилятор может выполнить так:
string null_book( "9-999-99999-9" ); // компилятор
пропускает конструктор
```

// копий

Но даже если компилятор обойдет вызов конструктора копий или перемещения, то он все равно должен существовать и быть доступен (не должен быть закрытым, например) в этой точке программы.

Упражнения раздела 13.1.1

Упражнение 13.1. Что такое конструктор копий? Когда он используется?

Упражнение 13.2. Объясните, почему следующее объявление недопустимо:

```
Sales_data::Sales_data( Sales_data rhs );
```

Упражнение 13.3. Объясните, что происходит при копировании объектов классов StrBlob и StrBlobPtr?

Упражнение 13.4. Предположим, класс Point имеет открытый конструктор копий. Укажите каждый случай использования конструктора копий в этом фрагменте кода:

```
Point global;
Point foo_bar( Point arg ) {
    Point local = arg, *heap = new Point( global );
    *heap = local;
    Point pa[ 4 ] = { local, *heap };
    return *heap;
}
```

Упражнение 13.5. Напишите с учетом следующего эскиза класса конструктор копий, копирующий все переменные-члены. Конструктор должен динамически резервировать новую строку (см. раздел 12.1.2) и копировать объект, на который указывает ps, а не сам указатель ps.

```
class HasPtr {
public:
    HasPtr( const std::string &s = std::string() ) :
        ps( new std::string( s ) ), i( 0 ) { }
private:
    std::string *ps;
    int i;
};
```



13.1.2. Оператор присвоения копии

Подобно тому, как класс контролирует инициализацию своих объектов, он контролирует также присваивание своих объектов:

```
Sales_data trans, accum;  
trans = accum; // использует оператор присвоения  
копии  
// класса Sales_data
```

Компилятор сам синтезирует оператор присвоения копии, если он не определен в классе явно.

Перегруженный оператор присвоения

Прежде чем перейти к синтезируемому оператору присвоения, необходимо ознакомиться с *перегрузкой операторов* (overloaded operator), подробно рассматриваемой в главе 14.

Перегруженные операторы — это функции, имена которых состоят из слова `operator` и символа определяемого оператора. Следовательно, оператор присвоения — это функция `operator=`. Подобно любой другой функции, у функции оператора есть тип возвращаемого значения и список параметров.

Параметрами перегруженного оператора являются его операнды. Некоторые операторы, например присвоение, должны быть определены, как функции-члены. Когда оператор является функцией-членом, левый операнд связан с неявным параметром `this` (см. раздел 7.1.2). Правый операнд бинарного оператора, такого как присвоение, передается как явный параметр. Оператор присвоения копии получает аргумент того же типа, что и класс:

```
class Foo {  
public:  
    Foo& operator=(const Foo&); // оператор присвоения  
    // ...  
};
```

Для совместимости с оператором присвоения встроенных типов (см. раздел 4.4) операторы присвоения обычно возвращают ссылку на свой левый операнд. Следует также заметить, что библиотека обычно требует от типов, хранимых в контейнере, наличия операторов присвоения, возвращающих ссылку на левый операнд.



Операторы присвоения обычно должны возвращать ссылку на свой левый операнд.

Синтезируемый оператор присвоения копии

Подобно конструктору копий, компилятор создает *синтезируемый оператор присвоения копии* (synthesized assignment operator) для класса, если в нем не определен собственный. Аналогично конструктору копий, у некоторых классов синтезируемый оператор присвоения копии не подразумевает присвоения (раздел 13.1.6). В противном случае он присваивает значение каждой нестатической переменной-члена правого объекта соответствующей переменной-члену левого объекта с использованием оператора присвоения копии типа этой переменной. Массивы присваиваются поэлементно. Синтезируемый оператор присвоения копии возвращает ссылку на свой левый операнд.

Например, следующий код эквивалентен синтезируемому оператору присвоения копии класса `Sales_data`:

```
// эквивалент синтезируемого оператора присвоения
// копии
Sales_data&
Sales_data::operator=(const Sales_data &rhs) {
    bookNo = rhs.bookNo;                                // вызов
string::operator=
    units_sold = rhs.units_sold;           // использует
встроенное присвоение
    int revenue = rhs.revenue;                // использует
встроенное
                                                // присвоение double
    return *this;                                // возвратить этот
объект
}
```

Упражнения раздела 13.1.2

Упражнение 13.6. Что такое оператор присвоения копии? Когда он используется? Что делает синтезируемый оператор присвоения копии? Когда он синтезируется?

Упражнение 13.7. Что произойдет при присвоении одного объекта класса `StrBlob` другому? Что произойдет при присвоении объектов класса `StrBlobPtr`?

Упражнение 13.8. Напишите оператор присвоения для класса `HasPtr` из упражнения 13.5 раздела 13.1.1. Подобно конструктору копий, данный оператор присвоения должен копировать объект, на который указывает указатель `ps`.

13.1.3. Деструктор

Действие деструктора противоположно действию конструктора: конструкторы инициализируют нестатические переменные-члены объекта, а также могут выполнять другие действия; деструкторы осуществляют все действия, необходимые для освобождения использованных объектом ресурсов и удаления нестатических переменных-членов объекта.

Деструктор — это функция-член с именем класса, предваряемым тильдой (~). У нее нет ни параметров, ни возвращаемого значения:

```
class Foo {  
public:  
    ~Foo(); // деструктор  
    // ...  
};
```

Поскольку деструктор не получает никаких параметров, он не может быть перегружен. Для каждого класса возможен только один деструктор.

Что делает деструктор

Подобно тому, как конструктор имеет часть инициализации и тело (см. раздел 7.5.1), деструктор имеет тело и часть удаления. В конструкторе переменные-члены инициализируются перед выполнением тела, а инициализация членов осуществляется в порядке их объявления в классе. В деструкторе сначала выполняется тело, а затем происходит удаление членов. Переменные-члены удаляются в порядке, обратном их инициализации.

Тело деструктора осуществляет все операции, которые разработчик класса считает необходимыми выполнить после использования объекта. Как правило, деструктор освобождает ресурсы объекта, зарезервированные на протяжении его существования.

У деструктора нет ничего похожего на список инициализации конструктора для контроля удаления переменных-членов; часть удаления

неявна. Происходящее при удалении переменной-члена зависит от его типа. Члены типа класса удаляются за счет выполнения его собственного деструктора. У встроенных типов нет деструкторов, поэтому для удаления членов встроенного типа не делается ничего.



Неявное удаление члена-указателя встроенного типа *не удаляет* объект, на который он указывает.

В отличие от обычных указателей, интеллектуальные указатели (см. раздел 12.1.1) являются классами и имеют деструкторы. Поэтому, в отличие от обычных указателей, члены, являющиеся интеллектуальными указателями, автоматически удаляются на фазе удаления.

Когда происходит вызов деструктора

Деструктор автоматически используется всякий раз, когда удаляется объект его типа.

- Переменные удаляются, когда выходят из области видимости.
- Переменные-члены объекта удаляются при удалении объекта, которому они принадлежат.
- Элементы в контейнере (будь то библиотечный контейнер или массив) удаляются при удалении контейнера.
- Динамически созданные объекты удаляются при применении оператора `delete` к указателю на объект (см. раздел 12.1.2).
- Временные объекты удаляются в конце выражения, в котором они были созданы.

Поскольку деструкторы выполняются автоматически, программы могут резервировать ресурсы и (обычно) не заботиться о том, когда они освобождаются.

Например, следующий фрагмент кода определяет четыре объекта класса `Sales_data`:

```
{ // новая область видимости
    // p и p2 указывают на динамически созданные
    // объекты Sales_data
    *p = new Sales_data;                                // p -
    // встроенный указатель
    auto p2 = make_shared<Sales_data>(); // p2 -
```

```

shared_ptr
    Sales_data item( *p);           // конструктор копий
копирует *p в item
    vector<Sales_data> vec; // локальный объект
    vec.push_back( *p2);          // копирует объект, на
который указывает p2
    delete p;                    // деструктор вызывается
для объекта, на
                                // который указывает p
} // выход из локальной области видимости;
деструктор вызывается
// для item, p2 и vec
// удаление p2 уменьшает его счетчик
пользователей; если значение
// счетчика дойдет до 0, объект освобождается
// удаление вектора vec удалит и его элементы

```

Каждый из этих объектов содержит член типа `string`, который резервирует динамическую память для содержания символов переменной-члена `bookNo`. Но единственная память, которой код должен управлять непосредственно, — это самостоятельно зарезервированный объект. Код непосредственно освобождает только динамически созданный объект, связанный с указателем `p`.

Другие объекты класса `Sales_data` автоматически удаляются при выходе из области видимости. По завершении блока `vec`, `p2` и `item` выходят из области видимости, это означает вызов деструкторов классов `vector`, `shared_ptr` и `Sales_data` для соответствующих объектов. Деструктор класса `vector` удалит элемент, помещенный в вектор `vec`. Деструктор класса `shared_ptr` осуществит декремент счетчика ссылок объекта, на который указывает указатель `p2`. В данном примере этот счетчик достигнет нуля, поэтому деструктор класса `shared_ptr` удалит объект класса `Sales_data`, зарезервированный с использованием указателя `p2`.

Во всех случаях деструктор класса `Sales_data` неявно удаляет переменную-член `bookNo`. Удаление переменной-члена `bookNo` запускает деструктор класса `string`, который освобождает память, используемую для хранения ISBN.



Когда из области видимости выходит ссылка или указатель на объект, деструктор *не выполняется*.

Синтезируемый деструктор

Компилятор определяет *синтезируемый деструктор* (synthesized destructor) для любого класса, который не определяет собственный деструктор. Подобно конструкторам копий и операторам присвоения копии, определение для некоторых классов синтезируемого деструктора предотвращает удаление объектов этого типа (раздел 13.1.6). В противном случае у синтезируемого деструктора будет пустое тело.

Например, синтезируемый деструктор класса `Sales_data` эквивалентен следующему:

```
class Sales_data {  
public:  
    // не делать ничего, кроме удаления переменных-  
    // членов,  
    // осуществляющего автоматически  
    ~Sales_data() {}  
    // другие члены как прежде  
};
```

Переменные-члены автоматически удаляются после выполнения (пустого) тела деструктора. В частности, деструктор класса `string` будет выполнен для освобождения памяти, используемой переменной-членом `bookNo`.

Важно понять, что само тело деструктора не удаляет переменные-члены непосредственно. Они удаляются в ходе неявной фазы удаления, которая следует за телом деструктора. Тело деструктора выполняется *в дополнение* к удалению членов, осуществляемому в ходе удаления объекта.

Упражнения раздела 13.1.3

Упражнение 13.9. Что такое деструктор? Что делает синтезируемый деструктор? Когда деструктор синтезируется?

Упражнение 13.10. Что произойдет при удалении объекта класса `StrBlob`? А класса `StrBlobPtr`?

Упражнение 13.11. Добавьте деструктор в класс HasPtr из предыдущих упражнений.

Упражнение 13.12. Сколько вызовов деструктора происходит в следующем фрагменте кода?

```
bool fcn( const Sales_data *trans, Sales_data accum)
{
    Sales_data item1(*trans), item2(accum);
    return item1.isbn() != item2.isbn();
}
```

Упражнение 13.13. Наилучший способ изучения функций-членов управления копированием и конструкторов — это определить простой класс с этими функциями-членами, каждая из которых выводит свое имя:

```
struct X {
    X() { std::cout << "X()" << std::endl; }
    X(const X&) { std::cout << "X(const X&)" << std::endl; }
};
```

Добавьте в структуру X оператор присвоения копии и деструктор, а затем напишите программу, использующую объекты класса X различными способами: передайте их как ссылочный и не ссылочный параметры; динамически зарезервируйте их; поместите в контейнеры и т.д. Изучайте вывод, пока не начнете хорошо понимать, когда и почему используется каждая функция-член управления копированием. По мере чтения вывода помните, что компилятор может обойти вызовы конструктора копий.



13.1.4. Правило три/пять

Как уже упоминалось, существуют три базовых функции, контролирующих копирование объектов класса: конструктор копий, оператор присвоения копии и деструктор. Кроме того, как будет продемонстрировано в разделе 13.6, по новому стандарту класс может также определить конструктор перемещения и оператор присваивания при перемещении.

Определять все эти функции не обязательно: вполне можно определить один или два из них, не определяя все. Эти функции можно считать модулями. Если нужен один, не обязательно определять их все.

Классы, нуждающиеся в деструкторах, нуждаются в копировании и присвоении

Вот эмпирическое правило, используемое при принятии решения о необходимости определения в классе собственных версий функций-членов управления копированием: сначала следует решить, нужен ли классу деструктор. Зачастую потребность в деструкторе более очевидна, чем потребность в операторе присвоения или конструкторе копий. Если класс нуждается в деструкторе, он почти наверняка нуждается также в конструкторе копий и операторе присвоения копии.

Используемый в упражнениях класс HasPtr отлично подойдет для примера (см. раздел 13.1.1). Этот класс резервирует динамическую память в конструкторе. Синтезируемый деструктор не будет удалять указатель-член. Поэтому данный класс должен определить деструктор для освобождения памяти, зарезервированной конструктором.

Хоть это и не очевидно, но согласно эмпирическому правилу класс HasPtr нуждается также в конструкторе копий и операторе присвоения копии.

Давайте посмотрим, что было бы, если бы у класса HasPtr был деструктор и синтезируемые версии конструктора копий и оператора присвоения копии:

```
class HasPtr {  
public:  
    HasPtr( const std::string &s = std::string() ):  
        ps( new std::string(s) ), i(0) { }  
    ~HasPtr() { delete ps; }  
    // ошибка: HasPtr нуждается в конструкторе копий и  
    // операторе  
    // присвоения копии  
    // другие члены, как прежде  
};
```

В этой версии класса зарезервированная в конструкторе память будет освобождена при удалении объекта класса HasPtr. К сожалению, здесь есть серьезная ошибка! Данная версия класса использует синтезируемые версии операторов копирования и присвоения. Эти функции копируют указатели-члены, а значит, несколько объектов класса HasPtr смогут указывать на ту же область памяти:

```
HasPtr f( HasPtr hp ) // HasPtr передан по значению,  
поэтому он
```

```

    // копируется
{
    HasPtr ret = hp; // копирует данный HasPtr
    // обработка ret
    return ret; // ret и hp удаляются
}

```

Когда функция `f()` завершает работу, объекты `hp` и `ret` удаляются и деструктор класса `HasPtr` выполняется для каждого из них. Этот деструктор удалит указатель-член и в объекте `ret`, и в объекте `hp`. Но эти объекты содержат одинаковое значение указателя. Код удалит тот же указатель дважды, что является серьезной ошибкой (см. раздел 12.1.2) с непредсказуемыми результатами.

Кроме того, вызывающая сторона функции `f()` может все еще использовать переданный ей объект:

```

HasPtr p("some values");
f(p); // по завершении f() память, на которую
указывает p, ps,
      // освобождается
HasPtr q(p); // теперь и p, и q указывают на
недопустимую память!

```

Память, на которую указывает указатель `p` (и `q`), больше недопустима. Она была возвращена операционной системе, когда был удален объект `hp` (или `ret`)!



Если класс нуждается в деструкторе, он почти наверняка нуждается также в операторе присвоения копии и конструкторе копий.

Классы, нуждающиеся в копировании, нуждаются также в присвоении, и наоборот

Хотя большинству классов требуется определить все функции-члены управления копированием (или ни один из них), у некоторых классов есть необходимость только в копировании или присвоении объектов, но нет никакой необходимости в деструкторе.

В качестве примера рассмотрим класс, присваивающий каждому своему объекту уникальный последовательный номер. Такому классу

нужен конструктор копий для создания нового уникального последовательного номера для создаваемого объекта. Этот конструктор копировал бы все остальные переменные-члены заданного объекта. Класс нуждался бы также в собственном операторе присвоения копии, чтобы избежать присвоения объекту слева последовательного номера. Однако у этого класса не было бы никакой потребности в деструкторе.

Этот пример иллюстрирует второе эмпирическому правило: если класс нуждается в конструкторе копий, то он почти наверняка нуждается в операторе присвоения копии, и наоборот, — если класс нуждается в операторе присвоения, то он почти наверняка нуждается также в конструкторе копий. Однако нужда в конструкторе копий или операторе присвоения копии не означает потребности в деструкторе.

Упражнения раздела 13.1.4

Упражнение 13.14. Предположим, что класс `numbered` имеет стандартный конструктор, создающий уникальный последовательный номер для каждого объекта, который хранится в переменной-члене `mysn`. Класс `numbered` использует синтезируемые функции-члены управления копированием и имеет следующую функцию:

```
void f( numbered s) { cout << s.mysn << endl; }
```

Какой вывод создаст следующий код?

```
numbered a, b = a, c = b;  
f( a); f( b); f( c);
```

Упражнение 13.15. Предположим, что у класса `numbered` есть конструктор копий, создающий новый последовательный номер. Изменит ли это вывод вызовов в предыдущем упражнении? Если да, то почему? Какой вывод получится?

Упражнение 13.16. Что если параметром функции `f()` будет `const numbered&`? Это изменяет вывод? Если да, то почему? Какой вывод получится?

Упражнение 13.17. Напишите версии класса `numbered` и функции `f()`, соответствующие трем предыдущим упражнениям, и проверьте правильность предсказания вывода.

13.1.5. Использование спецификатора = default



Используя спецификатор `= default`, можно явно указать компилятору на

необходимость создать синтезируемые версии функций-членов управления копированием (см. раздел 7.1.4):

```
class Sales_data {  
public:  
    // управление копированием; версии по умолчанию  
    Sales_data() = default;  
    Sales_data(const Sales_data&) = default;  
    Sales_data& operator=(const Sales_data &);  
    ~Sales_data() = default;  
    // другие члены как прежде  
};  
Sales_data&           Sales_data::operator=(const  
Sales_data&) = default;
```

Когда в объявлении функции-члена в теле класса использован спецификатор `= default`, синтезируемая функция неявно становится встраиваемой (как и любая другая функция-член, определенная в теле класса). Если синтезируемая функция-член класса не должна быть встраиваемой функцией, можно добавить часть `= default` в ее определение, как это было сделано в определении оператора присвоения копии.



Спецификатор `= default` можно использовать только для тех функций-членов, у которых есть синтезируемая версия (т.е. стандартный конструктор или функция-член управления копированием).

13.1.6. Предотвращение копирования

Рекомендуем

Большинство классов должно определить (явно или неявно) стандартный конструктор, конструктор копий и оператор присвоения копии.

Хотя большинство классов должно определять (и, как правило, определяет) конструктор копий и оператор присвоения копии, у некоторых классов нет реальной необходимости в этих функциях. В таких случаях класс должен быть определен так, чтобы предотвращать копирование и присвоение. Например, классы `iostream` предотвращают копирование, чтобы не позволять нескольким объектам писать или читать из того же буфера ввода-вывода. Казалось бы, предотвратить копирование можно, и не определяя функции-члены управления копированием. Но эта стратегия не сработает: если класс не определит эти функции, то компилятор синтезирует их сам.

Определение функции как удаленной



По новому стандарту можно предотвратить копирование, определив конструктор копий и оператор присвоения копии как *удаленные функции* (deleted function). Удаленной называется функция, которая была объявлена, но не может использована никаким другим способом. Чтобы определить функцию как удаленную, за списком ее параметров следует расположить часть = `delete`:

```
struct NoCopy {
    NoCopy() = default; // использовать синтезируемый
                         // стандартный
                         // конструктор
    NoCopy( const NoCopy&) = delete; // без
                                     // копирования
    NoCopy &operator=( const NoCopy&) = delete; // без
                                                 // присвоения
    ~NoCopy() = default; // используйте синтезируемый
```

```
деструктор
    // другие члены
};
```

Часть = `delete` указывает компилятору (и читателям кода), что эти функции-члены не определяются *преднамеренно*.

В отличие от части = `default`, часть = `delete` должна присутствовать в первом объявлении удаленной функции. Это различие согласуется со смыслом данных объявлений. Часть = `default` влияет только на то, какой код создает компилятор; следовательно, она необходима, только пока компилятор не создаст код. С другой стороны, компилятор должен знать, что функция удалена, и запретить ее использование другими функциями.

Также в отличие от части = `default`, часть = `delete` можно применить для любой функции (= `default` применима только к стандартному конструктору или функции-члену управления копированием, которую компилятор может синтезировать). Хотя изначально удаленные функции предназначались для подавления функций-членов управления копированием, они иногда применимы также для воздействия на процесс подбора функции.

Деструктор не должен быть удаленной функцией-членом

Следует заметить, что удалять деструктор нельзя. Если его удалить, то не будет никакого способа освободить объект этого типа. Компилятор не позволит определять переменные или создавать временные объекты типа, у которого удален деструктор. Кроме того, нельзя определять переменные или временные объекты класса, обладающего членом, у типа которого удален деструктор. Если у переменной-члена класса удален деструктор, то она не может быть освобождена. Если не может быть удалена переменная-член, не может быть удален и весь объект в целом.

Хотя определить переменные или переменные-члены таких типов нельзя, вполне можно динамически резервировать объекты с удаленным деструктором. Однако впоследствии их нельзя будет освободить:

```
struct NoDtor {
    NoDtor() = default; // использовать синтезируемый
                          // стандартный
                           // конструктор
    ~NoDtor() = delete; // нельзя удалять объекты типа
                        NoDtor
};
```

```
NoDtor nd; // ошибка: у NoDtor удаленный деструктор  
NoDtor *p = new NoDtor(); // ok: но нельзя удалить  
p  
delete p; // ошибка: у NoDtor удаленный деструктор
```



Невозможно определить объект или удалить указатель на динамически созданный объект типа с удаленным деструктором.

Функции-члены управления копированием могут быть синтезированы как удаленные

Как уже упоминалось, если не определены функции-члены управления копированием, компилятор определит их сам. Аналогично, если класс не определяет конструктор, компилятор синтезирует стандартный конструктор для этого класса сам (см. раздел 7.1.4). Для некоторых классов компилятор определяет эти синтезируемые функции-члены как удаленные.

- Синтезируемый деструктор определяется как удаленный, если у класса есть переменная-член, собственный деструктор которой удален или недоступен (например, `private`).
- Синтезируемый конструктор копий определяется как удаленный, если у класса есть переменная-член, собственный деструктор которой удален или недоступен. Он также будет удаленным, если у класса есть переменная-член с удаленным или недоступным деструктором.
- Синтезируемый оператор присвоения копии определяется как удаленный, если у класса есть переменная-член с удаленным или недоступным оператором присвоения копии, либо если у класса есть константный или ссылочный член.
- Синтезируемый стандартный конструктор определяется как удаленный, если у класса есть переменная-член с удаленным или недоступным деструктором; или имеется ссылочный член без внутриклассового инициализатора (см. раздел 2.6.1); или есть константная переменная-член, тип которой не определяет стандартный конструктор явно и не имеет внутриклассового инициализатора.

Короче говоря, эти правила означают, что если у класса есть переменная-член, которая не может быть стандартно создана, скопирована, присвоена или удалена, то соответствующая функция-член класса будет удаленной функцией.

Как ни удивительно, но переменная-член, класс которой имеет удаленный или недоступный деструктор, приводит к определению синтезируемого стандартного конструктора копий удаленным. Основание для этого правила в том, что без него возможно создание объектов, которые невозможно удалить.

Однако в том, что компилятор не будет синтезировать стандартный конструктор для класса со ссылочным или с константным членом, который не может быть создан стандартно, ничего удивительного нет. Нет ничего удивительного и в том, что класс с константным членом не может использовать синтезируемый оператор присвоения копии: в конце концов, этот оператор пытается присвоить значения всем членам классов. Однако присвоить новое значение константному объекту невозможно.

Хотя вполне возможно присвоить новое значение ссылке, это изменит значение объекта, на который она ссылается. Если бы оператор присвоения копии синтезировался для таких классов, то левый operand продолжил бы ссылаться на тот же объект, что и перед присвоением. Он не ссылался бы на тот же объект, что и правый operand. Поскольку это поведение вряд ли будет желательно, синтезируемый оператор присвоения копии определяется как удаленный, если у класса есть ссылочный член.

Как будет продемонстрировано в разделах 13.6.2, 15.7.2 и 19.6, есть и другие аспекты, в связи с которыми функции-члены копирования могут быть определены как удаленные.



Как правило, функции-члены управления копированием синтезируются как удаленные, когда невозможно скопировать, присвоить или удалить член класса.

Закрытые функции управления копированием

До появления нового стандарта классы предотвращали копирование, объявляя свой конструктор копий и оператор присвоения копии как закрытые (`private`):

```
class PrivateCopy {  
    // нет спецификатора доступа; следующие члены  
    // являются закрытыми  
    // по умолчанию; см. р. 7.2
```

```

    // функции управления копированием закрыты, а
потому недоступны
    // обычному пользовательскому коду
PrivateCopy( const PrivateCopy& );
PrivateCopy &operator=( const PrivateCopy& );
    // другие члены
public:
    PrivateCopy() = default;      // использовать
синтезируемый стандартный
                                // конструктор
    ~PrivateCopy(); // пользователи могут определять
объекты этого типа,
                                // но не копировать их
} ;

```

Поскольку деструктор является открытым (`public`), пользователи смогут определять объекты класса `PrivateCopy`. Но так как конструктор копий и оператор присвоения копии являются закрытыми (`private`), пользовательский код не сможет копировать такие объекты. Но дружественные классы и члены класса вполне могут создавать копии. Чтобы предотвратить копирование и друзьями, и членами класса, эти функции-члены объявляют закрытыми и не определяют их.

За одним исключением, рассматриваемым в разделе 15.2.1, вполне допустимо объявлять, но не определять функции-члены (см. раздел 6.1.2). Попытка использования неопределенной функции-члена приведет к отказу во время компоновки. При объявлении (без определения) закрытого конструктора копий можно предотвратить любую попытку скопировать объект класса: пользовательский код, пытающийся сделать копию, будет помечен как ошибочный во время компиляции; попытки копирования в функциях-членах или дружественных классах будут отмечены как ошибка во время редактирования.



Рекомендуем

Для классов, которые должны предотвратить копирование, следует определить собственный конструктор копий и оператор присвоения копии, используя часть `= delete` вместо объявления их закрытыми.

Упражнения раздела 13.1.6

Упражнение 13.18. Определите класс `Employee`, содержащий имя сотрудника и его уникальный идентификатор. Снабдите класс стандартным конструктором и конструктором, получающим строку, представляющую имя сотрудника. Каждый конструктор должен создавать уникальный идентификатор за счет приращения статической переменной-члена.

Упражнение 13.19. Должен ли класс `Employee` определить собственные версии функций-членов управления копированием? Если да, то почему? Если нет, то тоже почему? Реализуйте все члены управления копированием, в которых, на ваш взгляд, нуждается класс `Employee`.

Упражнение 13.20. Объясните, что происходит при копировании, присвоении и удалении объектов классов `TextQuery` и `QueryResult` из раздела 12.3.

Упражнение 13.21. Должны ли классы `TextQuery` и `QueryResult` определять собственные версии функций-членов управления копированием? Если да, то почему? Если нет, то почему? Реализуйте функции управления копированием, необходимые, по-вашему, в этих классах.



13.2. Управление копированием и ресурсами

Обычно классы, управляющие ресурсами, расположеными вне его, должны определять функции-члены управления копированием. Как упоминалось в разделе 13.6, такие классы нуждаются в деструкторах, освобождающих зарезервированные объектом ресурсы. Если класс нуждается в деструкторе, он почти наверняка нуждается также в конструкторе копий и операторе присвоения копии.

Чтобы определить эти функции-члены, сначала следует решить, что будет означать копирование объекта данного типа. Вообще, есть два способа: операцию копирования можно определить так, чтобы класс вел себя, как значение или как указатель.

У классов, которые ведут себя, как значения, есть собственное состояние. При копировании объекта как значения копия и оригинал независимы друг от друга. Внесенные в копию изменения никак не влияют на оригинал, и наоборот.

Классы, действующие как указатели, используют состояние совместно. При копировании объектов таких классов копии и оригиналы используют те же данные. Изменения, внесенные в копии, изменяют также оригинал, и наоборот.

Из использованных ранее библиотечных классов поведением, подобным значениям, обладали классы библиотечных контейнеров и класс `string`. Ничего удивительного, что класс `shared_ptr` демонстрирует поведение, подобное указателю, как и класс `StrBlob` (см. раздел 12.1.1). Типы ввода-вывода и класс `unique_ptr` не допускают ни копирования, ни присвоения, поэтому их поведение не похоже ни на значение, ни на указатель.

Чтобы проиллюстрировать эти два подхода, определим для используемого в упражнениях класса `HasPtr` функции-члены управления копированием. Сначала заставим класс действовать, как значение, а затем повторно реализуем его в версии, ведущей себя, как указатель.

У класса `HasPtr` есть два члена типа `int` и указатель на тип `string`. Обычно классы непосредственно копируют переменные-члены встроенного типа (кроме указателей); такие члены являются значениями, а следовательно, ведут себя обычно, как значения. Происходящее при

копировании указателя-члена определяет то, должно ли у такого класса, как `HasPtr`, быть поведение, подобное значению или указателю.

Упражнения раздела 13.2

Упражнение 13.22. Предположим, класс `HasPtr` должен вести себя, как значение. Таким образом, у каждого его объекта должна быть собственная копия строки, на которую указывает объект. Определения функций-членов управления копированием рассматривается в следующем разделе, но уже сейчас известно все необходимое для их реализации. Напишите конструктор копий класса `HasPtr` и оператор присвоения копии прежде, чем продолжите чтение.



13.2.1. Классы, действующие как значения

Для обеспечения поведения, подобного значению, у каждого объекта должна быть собственная копия ресурса, которым управляет класс. Это значит, что у каждого объекта класса `HasPtr` должна быть собственная копия строки, на которую указывает указатель `ps`. Для реализации поведения, подобного значению, классу `HasPtr` нужно следующее.

- Конструктор копий, который копирует строку, а не только указатель.
- Деструктор, освобождающий строку.
- Оператор присвоения копии, освобождающий строку существующего объекта и копирующий ее значение в строку правого операнда.

Вот подобная значению версия класса `HasPtr`:

```
class HasPtr {
public:
    HasPtr( const std::string &s = std::string() ):
        ps( new std::string(s)), i(0) { }
        // у каждого объекта класса HasPtr есть
собственный экземпляр строки,
        // на которую указывает указатель ps
    HasPtr( const HasPtr &p) :
        ps( new std::string( *p.ps)), i( p.i) { }
        HasPtr& operator=( const HasPtr & );
        ~HasPtr() { delete ps; }
private:
```

```
    std::string *ps;
    int i;
};
```

Класс достаточно прост, все, кроме оператора присвоения, определено в теле класса. Первый конструктор получает (необязательный) аргумент типа `string`. Он динамически резервирует собственную копию этой строки и сохраняет ее адрес в указателе `ps`. Конструктор копий также резервирует собственный экземпляр строки. Деструктор освобождает память, зарезервированную ее конструкторами, выполняя оператор `delete` для указателя-члена `ps`.

Подобный значению оператор присвоения копии

Обычно операторы присвоения объединяют действия деструктора и конструктора копий. Подобно деструктору, оператор присвоения освобождает ресурсы левого операнда. Подобно конструктору копий, оператор присвоения копирует данные из правого операнда. Однако критически важно, чтобы эти действия осуществлялись в правильной последовательности, даже если объект присваивается сам себе. Кроме того, по возможности следует писать собственные операторы присвоения так, чтобы они оставляли левый operand в корректном состоянии, иначе произойдет исключение (см. раздел 5.6.2).

В данном случае можно отработать случай присвоения самому себе (и сделать код устойчивым к исключению), осуществляя сначала копирование правого операнда. После копирования освобождается левый operand и указатель модифицируется так, чтобы он указывал на вновь зарезервированную строку:

```
HasPtr& HasPtr::operator=(const HasPtr &rhs) {
    auto newp = new string(*rhs.ps); // скопировать строку
    delete ps; // освободить прежнюю память
    ps = newp; // копировать данные из rhs в этот объект
    i = rhs.i;
    return *this; // возвратить этот объект
};
```

В этом операторе присвоения, безусловно, сначала выполняется работа конструктора: инициализатор `newp` идентичен инициализатору `ps` в конструкторе копий класса `HasPtr`. Затем, как в деструкторе, удаляется строка, на которую в настоящее время указывает указатель `ps`. Остается

только скопировать указатель на недавно созданную строку и значение типа `int` из `rhs` в этот объект.

Ключевая концепция. Операторы присвоения

Создавая оператор присвоения, следует учитывать два момента.

- Операторы присвоения должны работать правильно, если объект присваивается сам себе.
- Большинство операторов присвоения делят работу с деструктором и конструктором копий.

Шаблон разработки оператора присвоения подразумевает сначала копирование правого операнда в локальный временный объект. После копирования вполне безопасно удалить существующие члены левого операнда. Как только левый операнд будет освобожден, копировать данные из временного объекта в переменные-члены левого операнда.

Для иллюстрации важности принятия мер против присвоения самому себе рассмотрим, что случилось бы, выгляди оператор присвоения так:

```
// НЕПРАВИЛЬНЫЙ способ написания оператора присвоения!
HasPtr&
HasPtr::operator=(const HasPtr &rhs) {
    delete ps; // освобождает строку, на которую
    // указывает этот объект
    // если rhs и *this - тот же объект, произойдет
    // копирование удаленной
    // памяти!
    ps = new string(*(rhs.ps));
    i = rhs.i;
    return *this;
}
```

Если `rhs` и этот объект совпадают, удаление `ps` освободит строку, на которую указывают и `*this`, и `rhs`. При попытке копирования `*(rhs.ps)` в операторе `new` этот указатель указывает уже на недопустимую область памяти. Результат непредсказуем.



ВНИМАНИЕ

Для операторов присвоения критически важно работать правильно, даже если объект присваивается сам себе. Проще всего обеспечить это,

скопировав правый операнд перед удалением левого.

Упражнения раздела 13.2.1

Упражнение 13.23. Сравните функции-члены управления копированием, написанные для решения упражнений предыдущего раздела, с кодом, представленным здесь. Убедитесь, что понимаете различия, если таковые вообще есть, между вашим кодом и приведенным в книге.

Упражнение 13.24. Что будет, если в версии класса `HasPtr` данного раздела не определен деструктор? Что если не определен конструктор копий?

Упражнение 13.25. Предположим, необходимо определить версию класса `StrBlob`, действующего как значение. Предположим также, что необходимо продолжить использовать указатель `shared_ptr`, чтобы класс `StrBlobPtr` все еще мог использовать указатель `weak_ptr` для вектора. Переделанный класс будет нуждаться в конструкторе копий и операторе присвоения копии, но не в деструкторе. Объясните, что должны делать конструктор копий и оператор присвоения копий. Объясните, почему класс не нуждается в деструкторе.

Упражнение 13.26. Напишите собственную версию класса `StrBlob`, описанного в предыдущем упражнении.



13.2.2. Определение классов, действующих как указатели

Чтобы класс `HasPtr` действовал как указатель, конструктор копий и оператор присвоения копии должны копировать указатель-член, а не строку, на которую он указывает. Класс все еще будет нуждаться в собственном деструкторе, чтобы освободить память, зарезервированную получающим строку конструктором (см. раздел 13.6). Тем не менее в данном случае деструктор не может односторонне освободить связанную с ним строку. Это можно сделать только тогда, когда исчезнет последний указатель на строку.

Простейший способ заставить класс действовать как указатель — это использовать указатель `shared_ptr` для управления ресурсами в классе. При копировании (или присвоении) копируется (или присваивается)

указатель `shared_ptr`. Класс `shared_ptr` сам отслеживает количество пользователей, совместно использующих объект, на который он указывает. Когда пользователей больше нет, класс `shared_ptr` освобождает ресурс.

Но иногда управлять ресурсом следует непосредственно. В таких случаях может пригодиться *счетчик ссылок* (reference count) (см. раздел 12.1.1). Для демонстрации работы счетчика ссылок переопределим класс `HasPtr` так, чтобы обеспечить поведение, подобное указателю, но с использованием собственного счетчика ссылок.

Счетчики ссылок

Счетчик ссылок работает следующим образом.

- В дополнение к инициализации объекта каждый конструктор (кроме конструктора копий) создает счетчик. Этот счетчик отслеживает количество объектов, совместно использующих создаваемые данные. Сразу после создания объект только один, поэтому счетчик инициализируется значением 1.

- Конструктор копий не создает новый счетчик; он копирует переменные-члены переданного ему объекта, включая счетчик. Конструктор копий увеличивает значение этого совместно используемого счетчика, указывая на наличие еще одного пользователя данных этого объекта.

- Деструктор уменьшает значение счетчика, указывая, что стало на одного пользователя совместно используемых данных меньше. Если значение счетчика достигает нуля, деструктор удаляет данные.

- Оператор присвоения копии увеличивает счетчик правого операнда и уменьшает счетчик левого. Если счетчик левого операнда достигает нуля, значит, пользователей больше нет. В данном случае оператор присвоения копии должен удалить данные левого операнда.

Единственное затруднение — это решить, где разместить счетчик ссылок. Счетчик не может быть членом непосредственно класса объекта `HasPtr`. Чтобы убедиться почему, рассмотрим происходящее в следующем примере:

```
HasPtr p1("Hiya!");  
HasPtr p2(p1); // p1 и p2 указывают на ту же строку  
HasPtr p3(p1); // p1, p2 и p3 указывают на ту же строку
```

Если счетчик ссылок будет храниться в каждом объекте, то как модифицировать его правильно при создании объекта `p3`? Можно увеличить счетчик в объекте `p1` и скопировать счет в `p3`, но как

модифицировать счетчик в p2?

Один из способов решения этой проблемы в том, чтобы хранить счетчик в динамической памяти. При создании объекта резервируется также и новый счетчик. При копировании или присвоении объекта копируется и указатель на счетчик. Таким образом, и копия, и оригинал укажут на тот же счетчик.

Определение класса счетчика ссылок

Используя счетчик ссылок, можно написать подобную указателю версию класса HasPtr следующим образом:

```
class HasPtr {
public:
    // конструктор резервирует новую строку и новый
    // счетчик,
    // устанавливаемый в 1
    HasPtr( const std::string &s = std::string() ):
        ps( new std::string(s) ),      i( 0 ),      use( new
        std::size_t( 1 ) ) {}
    // конструктор копий копирует все три переменные-
    // члена и увеличивает
    // счетчик
    HasPtr( const HasPtr &p):
        ps( p.ps ),      i( p.i ),      use( p.use ) { ++*use; }
    HasPtr& operator=( const HasPtr& );
    ~HasPtr();
private:
    std::string *ps;
    int i;
    std::size_t      *use;      // член,      отслеживающий
    // количество объектов,
    // совместно использующих *ps
};
```

Здесь была добавлена новая переменная-член use, отслеживающая количество объектов, совместно использующих ту же строку. Получающий строку конструктор резервирует счетчик и инициализирует его значением 1, означающим наличие одного пользователя строкового члена класса этого объекта.

Функции-члены копирования подобного указателю класса

используют счетчик ссылок

При копировании или присвоении объектов класса HasPtr необходимо, чтобы копия и оригинал указывали на ту же строку. Таким образом, когда копируется объект класса HasPtr, копируется сам указатель ps, а не строка, на которую он указывает. При копировании увеличивается также счетчик, связанный с этой строкой.

Конструктор копий (определенный в классе) копирует все три члена переданного ему объекта класса HasPtr. Этот конструктор увеличивает также значение указателя-члена use, означая, что у строки, на которую указывают указатели ps и p, появился другой пользователь.

Деструктор не может безоговорочно удалить указатель ps, поскольку могли бы быть и другие объекты, указывающие на ту же область памяти. Вместо этого деструктор осуществляет декремент счетчика ссылок, означая, что строку совместно используют на один объект меньше. Если счетчик достигает нуля, деструктор освобождает память, на которую указывают указатели ps и use:

```
HasPtr::~HasPtr() {
    if ( --*use == 0) { // если счетчик ссылок достиг
0,
        delete ps;           // удалить строку
        delete use;          // и счетчик
    }
}
```

Оператор присвоения копии, как обычно, выполняет действия, общие для конструктора копий и деструктора. Таким образом, оператор присвоения должен увеличить счетчик правого операнда (действие конструктора копий) и декремент счетчика левого операнда, освобождая по мере необходимости используемую память (действие деструктора).

Кроме того, как обычно, оператор должен учитывать присвоение себя самому. Для этого инкремент счетчика rhs осуществляется прежде декремента счетчика в левом операнде.

Таким образом, если оба операнда являются тем же объектом, значение счетчика будет увеличено прежде проверки необходимости удаления указателей ps и use:

```
HasPtr& HasPtr::operator=(const HasPtr &rhs) {
    ++*rhs.use; // инкремент счетчика пользователей
правого операнда
    if ( --*use == 0) { // затем декремент счетчика
```

```

ЭТОГО ОБЪЕКТА
    delete ps;           // если никаких других
ПОЛЬЗОВАТЕЛЕЙ НЕТ
    delete use;          // освободить резервированные
ЧЛЕНЫ ЭТОГО ОБЪЕКТА
}
ps = rhs.ps;           // копировать данные из rhs в
ЭТОТ ОБЪЕКТ
i = rhs.i;
use = rhs.use;
return *this;          // возвратить этот объект
}

```

Упражнения раздела 13.2.2

Упражнение 13.27. Определите собственную версию класса HasPtr со счетчиком ссылок.

Упражнение 13.28. С учетом следующих классов реализуйте стандартный конструктор и необходимые функции-члены управления копированием.

(a) class TreeNode { private: std::string value; int count; TreeNode *left; TreeNode *right; };	(b) class BinStrTree { private: TreeNode *root; };
--	---

13.3. Функция swap()

Кроме функций-членов управления копированием, управляющие ресурсами классы зачастую определяют также функцию `swap()` (см. раздел 9.2.5). Определение функции `swap()` особенно важно для классов, которые планируется использовать с алгоритмами переупорядочивания элементов (см. раздел 10.2.3). Такие алгоритмы вызывают функцию `swap()` всякий раз, когда им нужен обмен двух элементов.

Если класс определяет собственную функцию `swap()`, алгоритм использует именно ее. В противном случае используется функция `swap()`, определенная библиотекой. Как обычно, хоть мы пока и не знаем, как реализуется функция `swap()`, концептуально несложно заметить, что

обмен двух объектов задействует копирование и два присвоения. Например, код обмена двух объектов подобного значению класса `HasPtr` (см. раздел 13.2.1) мог бы выглядеть так:

```
HasPtr temp = v1; // сделать временную копию
значения v1
v1 = v2;           // присвоить значение v2 объекту
v1
v2 = temp;         // присвоить сохраненное значение
v1 объекту v2
```

Этот код дважды копирует строку, которая первоначально принадлежала объекту `v1`: один раз, когда конструктор копий класса `HasPtr` копирует объект `v1` в объект `temp`, и второй раз, когда оператор присвоения присваивает объект `temp` объекту `v2`. Он также копирует строку, которая первоначально принадлежала объекту `v2`, когда объект `v2` присваивается объекту `v1`. Как уже упоминалось, копирование объекта, подобного значению класса `HasPtr`, резервирует новую строку и копирует строку, на которую указывает объект класса `HasPtr`.

В принципе ни одно из этих резервирований памяти не обязательно. Вместо того чтобы резервировать новые копии строки, можно было бы обменять указатели. Таким образом, имел бы смысл обменять два объекта класса `HasPtr` так, чтобы выполнить следующее:

```
string *temp = v1.ps; // создать временную копию
указателя в v1.ps
v1.ps = v2.ps;        // присвоить указатель v2.ps
указателю v1.ps
v2.ps = temp;          // присвоить сохраненный
указатель v1.ps
                           // указателю v2.ps
```

Написание собственной функции swap()

Переопределить стандартное поведение функции `swap()` можно, определив в классе ее собственную версию. Вот типичная реализация функции `swap()`:

```
class HasPtr {
    friend void swap(HasPtr&, HasPtr&);
    // другие члены, как в разделе 13.2.1
};
```

inline

```
void swap( HasPtr &lhs, HasPtr &rhs) {  
    using std::swap;  
    swap( lhs.ps, rhs.ps); // обмен указателями, а не  
    строковыми данными  
    swap( lhs.i, rhs.i); // обмен целочисленными  
    членами  
}
```

Все начинается с объявления функции `swap()`, дружественной, чтобы предоставить ей доступ к закрытым переменным-членам класса `HasPtr`. Поскольку функция `swap()` предназначена для оптимизации кода, определим ее как встраиваемую (см. раздел 6.5.2). Тело функции `swap()` вызывает функции `swap()` каждой из переменных-членов заданного объекта. В данном случае сначала обмениваются указатели, а затем целочисленные члены объектов, связанных с параметрами `rhs` и `lhs`.



В отличие от функций-членов управления копированием, функция `swap()` никогда не бывает обязательной. Однако ее определение может быть важно для оптимизации классов, резервирующих ресурсы.



Функции `swap()` должны вызвать функции `swap()`, а не `std::swap()`

В этом коде есть один важный нюанс: хотя в данном случае это не имеет значения, важно, чтобы функция `swap()` вызвала именно функцию `swap()`, а не `std::swap()`. В классе `HasPtr` переменные-члены имеют встроенные типы. Для встроенных типов нет специализированных версий функции `swap()`. В данном случае она вызывает библиотечную функцию `std::swap()`.

Но если класс имеет член, тип которого обладает собственной специализированной функцией `swap()`, то вызов функции `std::swap()` был бы ошибкой. Предположим, например, что есть другой класс по имени `Foo`, переменная-член `h` которого имеет тип `HasPtr`. Если не написать для класса `Foo` собственную версию функции `swap()`, то будет использована ее библиотечная версия. Как уже упоминалось, библиотечная функция

`swap()` осуществляет ненужное копирование строк, управляемых объектами класса `HasPtr`.

Ненужного копирования можно избежать, написав функцию `swap()` для класса `Foo`. Но версию функции `swap()` для класса `Foo` можно написать так:

```
void swap( Foo &lhs, Foo &rhs) {  
    // Ошибка: эта функция использует библиотечную  
версию  
    // функции swap(), а не версию класса HasPtr  
    std::swap( lhs.h, rhs.h); // обменять другие члены  
класса Foo  
}
```

Этот код нормально компилируется и выполняется. Однако никакого различия в производительности между этим кодом и просто использующим стандартную версию функции `swap()` не будет. Проблема в том, что здесь явно запрошен вызов библиотечной версии функции `swap()`. Однако нужна версия функции не из пространства имен `std`, а определенная в классе `HasPtr`.

Правильный способ написания функции `swap()` приведен ниже.

```
void swap( Foo &lhs, Foo &rhs) {  
    using std::swap;  
    swap( lhs.h, rhs.h); // использует функцию swap()  
класса HasPtr  
    // обменять другие члены класса Foo  
}
```

Все вызовы функции `swap()` обходятся без квалификаторов. Таким образом, каждый вызов должен выглядеть как `swap()`, а не `std::swap()`. По причинам, рассматриваемым в разделе 16.3, если есть специфическая для типа версия функции `swap()`, она будет лучшим соответствием, чем таковая из пространства имен `std`. В результате, если у типа есть специфическая версия функции `swap()`, вызов `swap()` будет распознан как относящийся к специфической версии. Если специфической для типа версии нет, то (с учетом объявления `using` для функции `swap()` в области видимости) при вызове `swap()` будет использована версия из пространства имен `std`.

У очень осторожных читателей может возникнуть вопрос: почему объявление `using` функции `swap()` не скрывает объявление функции `swap()` класса `HasPtr` (см. раздел 6.4.1). Причины, по которым работает

этот код, объясняются в разделе 18.2.3.

Использование функции *swap*() в операторах присвоения

Классы, определяющие функцию *swap*(), зачастую используют ее в определении собственного оператора присвоения. Эти операторы используют технологию, известную как *копия и обмен* (*copy and swap*). Она подразумевает *обмен* левого операнда с *копией* правого:

```
// обратите внимание: параметр rhs передается по значению. Это значит,
// что конструктор копии класса HasPtr копирует строку в правый
// operand rhs
HasPtr& HasPtr::operator=(HasPtr rhs) {
    // обменивает содержимое левого операнда с локальной переменной rhs
    swap(*this, rhs); // теперь rhs указывает на память, которую
                      // использовал этот объект
    return *this; // удаление rhs приводит к удалению
                  // указателя в rhs
}
```

В этой версии оператора присвоения параметр не является ссылкой. Вместо этого правый operand передается по значению. Таким образом, *rhs* — это копия правого операнда. Копирование объекта класса *HasPtr* приводит к резервированию новой копии строки данного объекта.

В теле оператора присвоения вызывается функция *swap*(), обменивающая переменные-члены *rhs* с таковыми в **this*. Этот вызов помещает указатель, который был в левом операнде, в *rhs*, и указатель, который был в *rhs*, — в **this*. Таким образом, после вызова функции *swap*() указатель-член в **this* указывает на недавно зарезервированную строку, являющуюся копией правого операнда.

По завершении оператора присвоения параметр *rhs* удаляется и выполняется деструктор класса *HasPtr*. Этот деструктор освобождает память, на которую теперь указывает *rhs*, освобождая таким образом память, на которую указывал левый operand.

В этой технологии интересен тот момент, что она автоматически отрабатывает присвоение себя себе и изначально устойчива к исключениям. Копирование правого operand'a до изменения левого

отрабатывает присвоение себя себе аналогично примененному в нашем первоначальном операторе присвоения (см. раздел 13.2.1). Это обеспечивает устойчивость к исключениям таким же образом, как и в оригинальном определении. Единственный код, способный передать исключение, — это оператор new в конструкторе копий. Если исключение произойдет, то это случится прежде, чем изменится левый операнд.



Операторы присвоения, использующие копию и обмен, автоматически устойчивы к исключениям и правильно отрабатывают присвоение себя себе.

Упражнения раздела 13.3

Упражнение 13.29. Объясните, почему вызов функции swap() в вызове swap(HasPtr&, HasPtr&) не приводит к бесконечной рекурсии.

Упражнение 13.30. Напишите и проверьте функцию swap() для подобной значению версии класса HasPtr. Снабдите свою функцию swap() оператором вывода примечания о ее выполнении.

Упражнение 13.31. Снабдите свой класс оператором < и определите вектор объектов класса HasPtr. Вставьте в вектор несколько элементов, а затем отсортируйте его (sort()). Обратите внимание на то, когда вызывается функция swap().

Упражнение 13.32. Получит ли преимущества подобная указателю версия класса HasPtr от определения собственной функции swap() ? Если да, то в чем это преимущество? Если нет, то почему?

13.4. Пример управления копированием

Несмотря на то что управление копированием обычно необходимо для классов, резервирующих ресурсы, управление ресурсами не единственная причина определения этих функций-членов. У некоторых классов может быть необходимость в учете или других действиях, выполняемых функциями управления копированием.

В качестве примера, нуждающегося в управлении копированием класса для учета, рассмотрим два класса, которые могли бы использоваться в приложении обработки почты. Эти классы, `Message` и `Folder`, представляют соответственно сообщение электронной (или другой) почты и каталог, в котором могло бы находиться это сообщение. Каждое сообщение может находиться в нескольких папках. Но может существовать только одна копия содержимого любого сообщения. Таким образом, если содержимое сообщения изменится, эти изменения отображаются при просмотре данного сообщения в любой из папок.

Для отслеживания того, какие сообщения в каких папках находятся, каждый объект класса `Message` будет хранить набор указателей на объекты класса `Folder`, в которых они присутствуют, а каждый объект класса `Folder` будет содержать набор указателей на его объекты класса `Message`. Эту конструкцию иллюстрирует рис. 13.1.

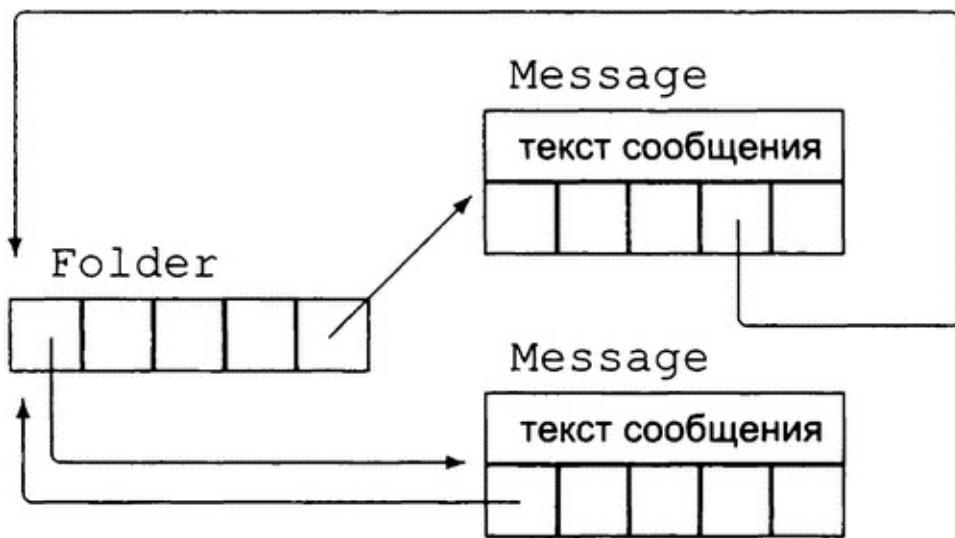


Рис. 13.1. Проект классов `Message` и `Folder`

Класс `Message` будет предоставлять функции `save()` и `remove()` для добавления и удаления сообщений из папки. Для создания нового объекта класса `Message` следует определить содержимое сообщения, но

не папку. Чтобы поместить сообщение в определенную папку, следует вызвать функцию `save()`.

После копирования сообщения копия и оригинал будут разными объектами класса `Message`, но оба сообщения должны присутствовать в том же самом наборе папок. Таким образом, копирование сообщения скопирует содержимое и набор указателей на папку. Он должен также добавить указатель на недавно созданный объект класса `Message` к каждому из этих объектов класса `Folder`.

После удаления сообщения объект класса `Message` больше не существует. Поэтому его удаление должно удалять указатели на этот объект класса `Message` из всех объектов класса `Folder`, которые содержали это сообщение.

Когда один объект класса `Message` присваивается другому, содержимое (`contents`) левого сообщения заменяется таковым правого. Следует также модифицировать набор папок, удалив левый объект класса `Message` из предыдущих объектов класса `Folder` и добавив в них правый.

Глядя на этот список операций, можно заметить, что и деструктор, и оператор присвоения копии должны удалять заданное сообщение из папок, которые указывают на него. Точно так же и конструктор копий, и оператор присвоения копии добавляют объект класса `Message` в заданный список объекта класса `Folder`. Для решения этих задач определим пару закрытых вспомогательных функций.

Рекомендуем

Оператор присвоения копии зачастую осуществляет ту же работу, которая необходима в конструкторе копий и деструкторе. В таких случаях эти действия обычно помещают в закрытые вспомогательные функции.

Класс `Folder` будет нуждаться в аналогичных функциях-членах управления копированием для добавления и удаления себя из хранящих их объектов класса `Message`.

Проектирование и реализацию класса `Folder` оставим читателю в качестве самостоятельного упражнения, но будем подразумевать, что у него есть функции-члены `addMsg()` и `remMsg()`, выполняющие все действия по добавлению и удалению заданного сообщения из набора сообщений указанной папки.

Класс Message

С учетом проекта выше можно написать класс Message следующим образом:

```
class Message {
    friend class Folder;
public:
    // папки неявно инициализируются пустым набором
    explicit Message( const std::string &str = "" ):
        contents(str) { }
    // функции управления копированием, контролирующие
    // указатели на
    // это сообщение
    Message( const Message& ); // конструктор
    // копий
    Message& operator=( const Message& ); // присвоение
    // копии
    ~Message(); // деструктор
    // добавить/удалить это сообщение из набора
    // сообщений папки
    void save(Folder&);
    void remove(Folder&);
private:
    std::string contents; // фактический текст
    // сообщения
    std::set<Folder*> folders; // папки, содержащие
    // это сообщение
    // вспомогательные функции, используемые
    // конструктором копий,
    // оператором присвоения и деструктором
    // добавить это сообщение в папки, на которые
    // указывает параметр
    void add_to_Folders( const Message& );
    // удалить это сообщение из каждой папки в folders
    void remove_from_Folders();
};
```

Класс определяет две переменные-члена: `contents` — для хранения текста сообщения и `folders` — для хранения указателей на объекты класса `Folder`, в которых присутствует данное сообщение. Получающий

строку конструктор копирует ее в переменную `contents` и (неявно) инициализирует переменную `folders` пустым набором. Поскольку у этого конструктора есть аргумент по умолчанию, он также является стандартным конструктором класса `Message` (см. раздел 7.5.1).

Функции-члены `save()` и `remove()`

Кроме функций управления копированием, у класса `Message` есть только две открытых функции-члена: `save()`, помещающая сообщение в данную папку, и `remove()`, извлекающая его:

```
void Message::save(Folder &f) {  
    folders.insert(&f); // добавить данную папку в  
    список папок  
    f.addMsg(this); // добавить данное сообщение в  
    набор сообщений  
}  
void Message::remove(Folder &f) {  
    folders.erase(&f); // удалить данную папку из  
    списка папок  
    f.remMsg(this); // удалить данное сообщение из  
    набора сообщений  
}
```

Чтобы сохранить (или удалить) сообщение, требуется модифицировать член `folders` класса `Message`. При сохранении сообщения сохраняется указатель на данный объект класса `Folder`; при удалении сообщения этот указатель удаляется.

Эти функции должны также модифицировать заданный объект класса `Folder`. Модификация этого объекта является задачей, контролируемой классом `Folder` при помощи функций-членов `addMsg()` и `remMsg()`, которые добавляют или удаляют указатель на данный объект класса `Message` соответственно.

Управление копированием класса `Message`

При копировании сообщения копия должна появляться в тех же папках, что и оригинальное сообщение. В результате необходимо перебрать набор указателей класса `Folder`, добавляя указатель на новое сообщение в каждую папку, на которую указывал оригинал сообщения. Для этого и конструктор копий, и оператор присвоения копии должны будут выполнять те же действия, поэтому определим функцию для этой общей

работы:

```
// добавить это сообщение в папки, на которые
указывает m
void Message::add_to_Folders( const Message &m) {
    for ( auto f : m.folders) // для каждой папки,
    // содержащей m,
        f->addMsg( this); // добавить указатель на
    это сообщение
                                            // в данную папку
}
```

Здесь происходит вызов функции `addMsg()` для каждого объекта класса `Folder` в `m.folders`. Функция `addMsg()` добавит указатель на этот объект класса `Message` в данный объект класса `Folder`.

Конструктор копий класса `Message` копирует переменные-члены данного объекта:

```
Message::Message( const Message &m):
contents( m.contents), folders( m.folders) {
    add_to_Folders( m); // добавить это сообщение в
папки, на которые
                                            // указывает m
}
```

А также вызывает функцию `add_to_Folders()`, чтобы добавить указатель на недавно созданный объект класса `Message` каждому объекту класса `Folder`, который содержит оригинал сообщения.

Деструктор класса `Message`

При удалении объекта класса `Message` следует удалить это сообщение из папок, которые указывают на него. Это общее действие с оператором присвоения копии, поэтому определим для этого общую функцию:

```
// удалить это сообщение из соответствующих папок
void Message::remove_from_Folders() {
    for ( auto f : folders) // для каждого указателя в
folders
        f->remMsg( this); // удалить это сообщение из
данной папки
}
```

Реализация функции `remove_from_Folders()` подобна таковой у функции `add_to_Folders()`, за исключением того, что она использует

функцию `remMsg()` для удаления текущего сообщения.

При наличии функции `remove_from_Folders()` написать деструктор несложно:

```
Message::~Message() {
    remove_from_Folders();
}
```

Вызов функции `remove_from_Folders()` гарантирует отсутствие у объектов класса `Folder` указателей на удаленный объект класса `Message`. Компилятор автоматически вызывает деструктор класса `string` для освобождения объекта `contents`, а деструктор класса `set` освобождает память, используемую элементами набора.

Оператор присвоения копии класса Message

Как обычно, оператор присвоения и оператор присвоения копии класса `Folder` должны выполнять действия конструктора копий и деструктора. Как всегда, крайне важно структурировать свой код так, чтобы он выполнялся правильно, даже если операнды слева и справа — тот же объект.

В данном случае защита против присвоения самому себе осуществляется за счет удаления указателей на это сообщение из папок левого операнда прежде, чем вставить указатели в папки правого операнда:

```
Messages Message::operator=( const Message &rhs) {
    // отработать присвоение себе самому, удаляя
    // указатели прежде вставки
    remove_from_Folders();           // обновить существующие
    // папки
    contents = rhs.contents;        // копировать содержимое
    // сообщения из rhs
    folders = rhs.folders;          // копировать указатели
    // Folder из rhs
    add_to_Folders(rhs);            // добавить это сообщение
    // к данным папкам
    return *this;
}
```

Если левый и правый операнды — тот же объект, то у них тот же адрес. Если вызвать функцию `remove_from_Folders()` после вызова функции `add_to_Folders()`, это сообщение будет удалено изо всех соответствующих ему папок.

Функция swap() класса Message

Библиотека определяет версии функции swap() для классов string и set (см. раздел 9.2.5). В результате класс Message извлечет пользу из определения собственной версии функции swap(). При определении специфической для класса Message версии функции swap() можно избежать лишних копирований членов contents и folders.

Но наша функция swap() должна также управлять указателями Folder, которые указывают на обмениваемые сообщения. После такого вызова, как swap(m1, m2), указатели Folder, указывающие на объект m1 , должны теперь указать на объект m2 , и наоборот.

Для управления указателями Folder осуществляются два прохода по всем элементам folders . Первый проход удалит сообщения из соответствующих папок. Затем вызов функции swap() совершил обмен переменных-членов. Второй проход по элементам folders добавляет указатели на обмениваемые сообщения:

```
void swap( Message &lhs, Message &rhs) {
    using std::swap; // в данном случае не
обязательно, но привычка
                                // хорошая
    // удалить указатели на каждое сообщение из их
(оригинальных) папок
    for (auto f: lhs.folders)
        f->remMsg( &lhs );
    for (auto f: rhs.folders)
        f->remMsg( &rhs ); // обмен наборов указателей
contents и folders
    swap( lhs.folders, rhs.folders ); // использует
swap( set&, set& )
    swap( lhs.contents, rhs.contents ); // swap( string&,
string& )
    // добавляет указатели на каждое сообщение в их
(новые) папки
    for (auto f: lhs.folders)
        f->addMsg( &lhs );
    for (auto f: rhs.folders)
        f->addMsg( &rhs );
}
```

Упражнения раздела 13.4

Упражнение 13.33. Почему параметр функций-членов `save()` и `remove()` класса `Message` имеет тип `Folder&`? Почему этот параметр не определен как `Folder` или `const Folder&`?

Упражнение 13.34. Напишите класс `Message`, как описано в этом разделе.

Упражнение 13.35. Что случилось бы, используй класс `Message` синтезируемые версии функций-членов управления копированием?

Упражнение 13.36. Разработайте и реализуйте соответствующий класс `Folder`. Этот класс должен содержать набор указателей на сообщения в той папке.

Упражнение 13.37. Добавьте в класс `Message` функции-члены удаления и вставки заданного `Folder*` в `folders`. Эти члены аналогичны функциям-членам `addMsg()` и `remMsg()` класса `Folder`.

Упражнение 13.38. Для определения оператора присвоения класса `Message` не использовалась технология копирования и обмена. Почему, по вашему?



13.5. Классы, управляющие динамической памятью

Некоторые классы должны резервировать переменный объем памяти во время выполнения. Такие классы зачастую способны (а если способны, то обычно обязаны) использовать библиотечный контейнер для хранения данных. Например, для хранения своих элементов класс `StrBlob` использует вектор.

Но эта стратегия срабатывает не для каждого класса; некоторые из них должны самостоятельно резервировать память. Обычно такие классы определяют собственные функции-члены управления копированием, чтобы управлять памятью, которую они резервируют.

В качестве примера реализуем упрощенную версию библиотечного класса `vector`. Кроме прочих упрощений, этот класс не будет шаблоном, он сможет хранить только строки. Поэтому назовем этот класс `StrVec`.

Проект класса `StrVec`

Как уже упоминалось, класс `vector` хранит свои элементы в непрерывном хранилище. Для повышения производительности класс `vector` предварительно резервирует хранилище, размер которого превосходит необходимое количество элементов (см. раздел 9.4). Каждая добавляющая элементы функция-член вектора проверяет наличие доступного пространства для следующего элемента. Если это так, элемент размещается в следующей доступной ячейке. Если места нет, вектор пересоздается: он резервирует новое пространство, перемещает в него существующие элементы, освобождает прежнее пространство и добавляет новый элемент.

Подобную стратегию и будем использовать в классе `StrVec`. Для получения пустой памяти используем класс `allocator` (см. раздел 12.2.2). Поскольку резервируемая классом `allocator` память пуста, используем его функцию-член `construct()` для создания объектов в этом пространстве, когда необходимо добавить новый элемент. Точно так же при удалении элемента используем его функцию-член `destroy()`.

У каждого объекта класса `StrVec` будет три указателя на пространство, используемое для хранения его элементов:

- указатель `elements` на первый элемент в зарезервированной памяти;

- указатель `first_free` на следующий элемент после фактически последнего;
- указатель `cap` на следующий элемент после конца зарезервированной памяти.

Значение этих указателей представлено на рис. 13.2.

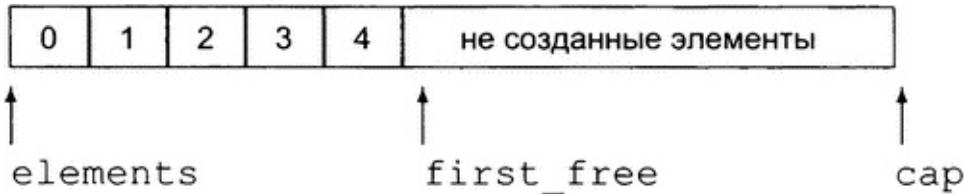


Рис. 13.2. Стратегия резервирования памяти класса `StrVec`

Кроме этих указателей, класс `StrVec` будет иметь переменную-член `alloc` типа `allocator<string>` для резервирования памяти, используемой классом `StrVec`. У класса также будет четыре вспомогательных функции.

- Функция `alloc_n_copy()` будет резервировать пространство и копировать заданный диапазон элементов.
- Функция `free()` будет удалять созданные элементы и освобождать пространство.
- Функция `chk_n_alloc()` будет гарантировать наличие достаточного места для добавления по крайней мере еще одного элемента в вектор `StrVec`. Если места для следующего элемента нет, то функция `chk_n_alloc()` вызовет функцию `reallocate()` для резервирования большего пространства.
- Функция `reallocate()` будет пересоздавать вектор `StrVec`, когда прежнее пространство окажется исчерпано.

Хотя основное внимание удалено реализации, определим также несколько членов из интерфейса класса `vector`.

Определение класса `StrVec`

Теперь, сделав набросок реализации, можно определить класс `StrVec`:

```
// упрощенная реализация стратегии резервирования
// памяти для подобного
// вектору класса
class StrVec {
public:
    StrVec(): // член allocator инициализируется по
    умолчанию
```

```

        elements( nullptr ),      first_free( nullptr ),
cap( nullptr ) { }
        StrVec( const StrVec& );           // конструктор
КОПИЙ
        StrVec &operator=( const StrVec& ); // присвоение
КОПИИ
        ~StrVec();                      // деструктор
        void push_back( const std::string& ); // копирует
ЭЛЕМЕНТ
        size_t size() const { return first_free - elements; }
        size_t capacity() const { return cap - elements; }
        std::string *begin() const { return elements; }
        std::string *end() const { return first_free; }
        // ...
private:
        std::allocator<std::string> alloc; // резервирует
ЭЛЕМЕНТЫ
        // используется функциями, которые добавляют
ЭЛЕМЕНТЫ в StrVec
        void chk_n_alloc()
        { if (size() == capacity()) reallocate(); }
        // вспомогательные члены, используемые
КОНСТРУКТОРОМ КОПИЙ,
        // оператором присвоения и деструктором
        std::pair<std::string*, std::string*> alloc_n_copy
        (const std::string*, const std::string* );
        void free(); // удаляет элементы и освобождает
пространство
        void reallocate(); // резервирует больше места и
копирует
                                // существующие элементы
        std::string *elements; // указатель на первый
ЭЛЕМЕНТ МАССИВА
        std::string *first_free; // указатель на первый
СВОБОДНЫЙ
                                // элемент массива
        std::string *cap; // указатель на следующий
ЭЛЕМЕНТ ПОСЛЕ

```

```
// конца массива  
};
```

Тело класса определяет некоторые из своих членов.

- Стандартный конструктор (неявно) инициализирует по умолчанию переменную-член `alloc` и (явно) инициализирует указатели как `nullptr`, означая, что никаких элементов нет.
- Функция-член `size()` возвращает количество фактически используемых элементов, соответствует значению `first_free - elements`.
- Функция-член `capacity()` возвращает количество элементов, которые может содержать объект класса `StrVec`, соответствует значению `cap - elements`.
- Функция-член `chk_n_alloc()` приводит к пересозданию объекта класса `StrVec`, когда больше нет места для добавления следующего элемента. Это происходит при `cap == first_free`.
- Функции-члены `begin()` и `end()` возвращают указатели на первый (т.е. `elements`) и следующий после последнего существующего элемент (т.е. `first_free`) соответственно.

Использование функции-члена `construct()`

Функция `push_back()` вызывает функцию `chk_n_alloc()`, чтобы удостовериться в наличии места для элемента. В случае необходимости функция `chk_n_alloc()` вызовет функцию `reallocate()`. После вызова функции `chk_n_alloc()` функция `push_back()` знает, что место для нового элемента есть. Она запрашивает свой член класса `allocator` создать новый последний элемент:

```
void StrVec::push_back(const string& s) {  
    chk_n_alloc(); // удостовериться в наличии места  
    // для другого элемента  
    // создать копию s в элементе, на который  
    // указывает first_free  
    alloc.construct(first_free++, s);  
}
```

При использовании класса `allocator` для резервирования памяти следует помнить, что память резервируется пустой (см. раздел 12.2.2). Чтобы использовать эту память, следует вызвать функцию `construct()`, которая создаст объект в этой памяти. Первый аргумент функции `construct()` — это указатель на пустое пространство,

зарезервированное вызовом функции `allocate()`. Остальные аргументы определяют, какой конструктор использовать при создании объекта в этом пространстве. В данном случае есть только один дополнительный аргумент типа `string`, поэтому этот вызов использует строковый конструктор копий.

Следует заметить, что вызов функции `construct()` осуществляет приращение указателя `first_free`, чтобы он снова указывал на элемент, который предстоит создать. Поскольку используется постфиксный инкремент (см. раздел 4.5), этот вызов создает объект в текущей позиции указателя `first_free`, а инкремент переводит его на следующий пустой элемент.

Функция-член `alloc_n_copy()`

Функция-член `alloc_n_copy()` вызывается при копировании или присвоении объекта класса `StrVec`. У класса `StrVec` будет подобное значению поведение (см. раздел 13.2.1), как у вектора; при копировании или присвоении объекта класса `StrVec` необходимо зарезервировать независимую память и скопировать элементы из оригинала в новый объект класса `StrVec`.

Функция-член `alloc_n_copy()` будет резервировать достаточно места для содержания заданного диапазона элементов, а затем скопировать эти элементы во вновь созданное пространство. Эта функция возвращает значение типа `pair` (см. раздел 11.2.3), переменные-члены которого являются указателем на начало нового пространства и следующую позицию после последнего скопированного элемента:

```
pair<string*, string*>
StrVec::alloc_n_copy( const string *b, const string
*e) {
    // резервировать пространство для содержания
    // элементов диапазона
    auto data = alloc.allocate( e - b);
    // инициализировать и возвратить пару, созданную
    // из данных,
    // возвращенных функцией uninitialized_copy()
    return { data, uninitialized_copy( b, e, data) };
}
```

Функция `alloc_n_copy()` вычисляет объем резервируемого пространства, вычитая указатель на первый элемент из указателя на

следующий после последнего. Зарезервировав память, функция создает в ней копии заданных элементов.

Копирование осуществляется в операторе `return` при списочной инициализации возвращаемого значения (см. раздел 6.3.2). Указатель-член `first` возвращенной пары указывает на начало зарезервированной памяти; значение для указателя-члена `second` возвращается функцией `uninitialized_copy()` (см. раздел 12.2.2). Это значение будет указателем на следующий элемент после последнего созданного элемента.

Функция-член `free()`

У функции-члена `free()` две обязанности: она должна удалить элементы, а затем освободить пространство, зарезервированное объектом класса `StrVec`. Цикл `for` вызывает функцию `destroy()` класса `allocator`, перебирая элементы в обратном порядке, начиная с последнего существующего элемента и заканчивая первым:

```
void StrVec::free() {
    // нельзя освободить 0 указателей;
    // если элемент нулевой - не делать ничего
    if (elements) {
        // удалить прежние элементы в обратном порядке
        for (auto p = first_free; p != elements; /* пусто
*) {
            alloc.destroy( --p );
            alloc.deallocate(elements, cap - elements);
        }
    }
}
```

Функция `destroy()` запускает деструктор класса `string`. Деструктор класса `string` освобождает память, занятую самой строкой.

Как только элементы будут удалены, освобождается пространство, зарезервированное классом `StrVec` при вызове функции `deallocate()`. Указатель, передаваемый функции `deallocate()`, должен быть именно тем, который ранее создал вызов функции `allocate()`. Поэтому перед вызовом функции `deallocate()` сначала проверяется, тот ли это `elements`, а не нулевой.

Функции-члены управления копированием

При наличии функций-членов `alloc_n_copy()` и `free()` функции-члены управления копированием нашего класса очень просты.

```

StrVec::StrVec( const StrVec &s) {
    // вызов функции alloc_n_copy() для резервирования
    // количества
    // элементов как в s
    auto newdata = alloc_n_copy( s.begin(), s.end() );
    elements = newdata.first;
    first_free = cap = newdata.second;
}

```

Конструктор копий вызывает функцию `alloc_n_copy()`, а затем присваивает результат вызова переменным-членам. Возвращаемое значение функции `alloc_n_copy()` является парой указателей. Первый указатель указывает на первый созданный элемент, а второй — на следующий после последнего созданного. Поскольку функция `alloc_n_copy()` резервирует пространство для точно такого количества элементов, которое было задано, указатель `cap` также указывает только на следующий после последнего созданного.

Деструктор вызывает функцию `free()`:

```
StrVec::~StrVec() { free(); }
```

Оператор присвоения копии вызывает функцию `alloc_n_copy()` прежде, чем освободить существующие элементы. Это защищает от копирования в себя самого:

```

StrVec &StrVec::operator=( const StrVec &rhs) {
    // вызов alloc_n_copy() для резервирования точно
    // такого количества
    // элементов, как в rhs
    auto data = alloc_n_copy( rhs.begin(), rhs.end() );
    free();
    elements = data.first;
    first_free = cap = data.second;
    return *this;
}

```

Подобно конструктору копий, оператор присвоения копии использует значения, возвращенные функцией `alloc_n_copy()`, для инициализации своих указателей.



Перемещение, а не копирование элементов при резервировании

Прежде чем приступить к функции `realloc()`, следует обдумать то, что она должна делать:

- зарезервировать память для нового, большего массива строк;
- заполнить первую часть этого пространства существующими элементами;
- удалить элементы в существующей памяти и освободить ее.

Глядя на этот список, можно заметить, что пересоздание объекта класса `StrVec` влечет за собой копирование каждой строки из прежнего объекта `StrVec` в новый. Даже без подробностей реализации класса `string` известно, что строки ведут себя подобно значению. После копирования новая строка и оригинальная независимы друг от друга. Изменения, внесенные в оригинал, не распространяются на копию, и наоборот.

Поскольку строки действуют, как значения, можно сделать вывод, что у каждой строки должна быть собственная копия составляющих ее символов. Копирование строки должно резервировать память для этих символов, а удаление строки должно освободить используемую ею память.

Копирование строки подразумевает копирование данных, поскольку обычно после копирования строки у нее будет два пользователя. Но когда функция `realloc()` копирует строки объекта класса `StrVec`, у этих строк будет только один пользователь. Как только копирование элементов из прежнего пространства в новое завершается, исходные строки немедленно удаляются.

Копирование данных этих строк не нужно. Производительность класса `StrVec` будет значительно выше, если удастся избежать дополнительных затрат на резервирование и освобождение строк при каждом его пересоздании.



Конструктор перемещения и функция `std::move()`

Копирования строки можно избежать при помощи двух средств, введенных новой библиотекой. Во-первых, некоторые из библиотечных классов, включая класс `string`, определяют так называемые *конструкторы перемещения* (*move constructor*). Подробности работы конструктора перемещения класса `string` (равно как и все остальные подробности его реализации) не раскрываются. Однако общеизвестно, что конструкторы перемещения обычно "перемещают" ресурсы из заданного объекта в создаваемый. Библиотека гарантирует также то, что

"перемещенная" строка останется в допустимом состоянии. В случае класса `string` можно предположить, что у каждого его объекта есть указатель на массив типа `char`. По-видимому, конструктор перемещения класса `string` копирует указатель вместо резервирования нового пространства и копирования символов.

Второе доступное для использования средство — это библиотечная функция `move()`, определенная в заголовке `utility`. Есть два важных момента, которые следует знать о функции `move()`. Во-первых, по причинам, рассматриваемым в разделе 13.6.1, когда функция `reallocate()` создает строки в новой области памяти, она должна вызвать функцию `move()`, чтобы сообщить о необходимости использования конструктора перемещения класса `string`. Если пропустить вызов функции `move()`, то будет использован конструктор копий класса `string`. Во-вторых, по причинам, рассматриваемым в разделе 18.2.3, объявление `using` (см. раздел 3.1) для функции `move()` обычно не предоставляется. Когда используется функция `move()`, вызывается функция `std::move()`, а не `move()`.

Функция-член `reallocate()`

Используя эту информацию, можно написать собственную функцию `reallocate()`. Сначала вызовем функцию `allocate()`, чтобы зарезервировать новое пространство. При каждом пересоздании объекта класса `StrVec` будем удваивать его емкость. Если вектор `StrVec` пуст, резервируем место для одного элемента:

```
void StrVec::reallocate() {
    // будем резервировать вдвое больше элементов, чем
    // текущий размер
    auto newcapacity = size() ? 2 * size() : 1;
    // резервировать новую память
    auto newdata = alloc.allocate(newcapacity);
    // переместить данные из прежней памяти в новую
    auto dest = newdata; // указывает на следующую
    // свободную позицию в
    // новом массиве
    auto elem = elements; // указывает на следующий
    // элемент в старом
    // массиве
    for (size_t i = 0; i != size(); ++i)
```

```

    alloc.construct( dest++, std::move( *elem++ ) );
    free( ); // освобождает старое пространство после
перемещения
        // элементов
    // обновить структуру данных, чтобы указать на
новые элементы
elements = newdata;
first_free = dest;
cap = elements + newcapacity;
}

```

Цикл `for` перебирает существующие элементы и создает соответствующие элементы в новом пространстве. Указатель `dest` используется для указания на область памяти, в которой создается новая строка, а указатель `elem` — для указания на элемент в оригинальном массиве. Для перемещения указателей `dest` и `elem` на следующий элемент этих двух массивов используем постфиксный инкремент.

Второй аргумент в вызове функции `construct()` (т.е. аргумент, определяющий используемый конструктор (см. раздел 12.2.2)) является значением, возвращенным функцией `move()`. Вызов функции `move()` возвращает результат, заставляющий функцию `construct()` использовать конструктор перемещения класса `string`. Поскольку используется конструктор перемещения, управляемая память строки не будет скопирована. Вместо этого каждая создаваемая строка получит в собственность область памяти из строки, на которую указывает указатель `elem`.

После перемещения элементов происходит вызов функции `free()` для удаления прежних элементов и освобождения памяти, которую данный вектор `StrVec` использовал перед вызовом функции `reallocate()`. Сами строки больше не управляют памятью, в которой они располагались; ответственность за их данные была передана элементам нового вектора `StrVec`. Нам неизвестно содержимое строк в памяти прежнего вектора `StrVec`, но нам гарантирована безопасность запуска деструктора класса `string` для этих объектов.

Остается только обновить указатели адресами вновь созданного и инициализированного массива. Указатели `first_free` и `cap` обозначают элемент следующий после последнего созданного и следующий после последнего зарезервированного соответственно.

Упражнения раздела 13.5

Упражнение 13.39. Напишите собственную версию класса `StrVec`, включая функции `reserve()`, `capacity()` (см. раздел 9.4) и `resize()` (см. раздел 9.3.5).

Упражнение 13.40. Добавьте в класс `StrVec` конструктор, получающий аргумент типа `initializer_list<string>`.

Упражнение 13.41. Почему в вызове функции `construct()` в функции `push_back()` был использован постфиксный инкремент? Что случилось бы при использовании префиксного инкремента?

Упражнение 13.42. Проверьте свой класс `StrVec`, использовав его в классах `TextQuery` и `QueryResult` (см. раздел 12.3) вместо вектора `vector<string>`.

Упражнение 13.43. Перепишите функцию-член `free()` так, чтобы для удаления элементов вместо цикла `for` использовалась функция `for_each()` и лямбда-выражение (см. раздел 10.3.2). Какую реализацию вы предпочитаете и почему?

Упражнение 13.44. Напишите класс по имени `String`, являющийся упрощенной версией библиотечного класса `string`. У вашего класса должен быть по крайней мере стандартный конструктор и конструктор, получающий указатель на строку в стиле С. Примените для резервирования используемой классом `String` памяти класс `allocator`.

13.6. Перемещение объектов

Одной из главных особенностей нового стандарта является способность перемещать объект, а не копировать. Как упоминалось в разделе 13.1.1, копирование осуществляется при многих обстоятельствах. При некоторых из них объект разрушается немедленно после копирования. В этих случаях перемещение объекта вместо копирования способно обеспечить существенное увеличение производительности.

Как было продемонстрировано только что, наш класс `StrVec` — хороший пример лишнего копирования. Во время пересоздания нет никакой необходимости в копировании элементов из старой памяти в новую, лучше перемещение. Вторая причина предпочтеть перемещение копированию — это такие классы как `unique_ptr` и классы ввода-вывода. У этих классов есть ресурс (такой как указатель или буфер ввода-вывода), который не допускает совместного использования. Следовательно, объекты этих типов не могут быть скопированы, но могут быть перемещены.

В прежних версиях языка не было непосредственного способа перемещения объекта. Копию приходилось делать, даже если в этом не было никакой потребности. Когда объекты велики или когда они требуют резервирования памяти (например, строки), бесполезное копирование может обойтись очень дорого. Точно так же в предыдущих версиях библиотеки классы хранимых в контейнере объектов должны были допускать копирование. По новому стандарту в контейнерах можно хранить объекты типов, которые не допускают копирования, но могут быть перемещены.



Контейнеры библиотечных типов, классы `string` и `shared_ptr` поддерживают как перемещение, так и копирование. Классы ввода-вывода и класс `unique_ptr` допускают перемещение, но не копирование.



13.6.1. Ссылки на r-значение



Для обеспечения операции пересылки, новый стандарт вводит новый вид ссылок — ссылки на r-значение. *Ссылка на r-значение* (r-value reference) — это ссылка, которая должна быть связана с r-значением. Ссылку на r-значение получают с использованием символа `&&`, а не `&`. Как будет продемонстрировано далее, у ссылок на r-значение есть важное свойство — они могут быть связаны только с тем объектом, который будет удален. В результате можно "перемещать" ресурсы от ссылки на r-значение в другой объект.

Напомним, что l- и r-значение — свойства выражения (см. раздел 4.1.1). Некоторые выражения возвращают или требуют l-значений; другие возвращают или требуют r-значений. Как правило, выражение l-значения относится к идентификатору объекта, тогда как выражение r-значения — к значению объекта.

Как и любая ссылка, ссылка на r-значение — это только другое имя для объекта. Как известно, нельзя связать обычные ссылки (которые далее будем называть *ссылками на l-значение* (l-value reference), чтобы отличить их от ссылок на r-значения) с выражениями, требующими преобразования, с литералами и с выражениями, которые возвращают r-значение (см. раздел 2.3.1). У ссылок на r-значение противоположные свойства привязки: можно связать ссылку на r-значение с выражениями, приведенными выше, но нельзя непосредственно связать ссылку на r-значение с l-значением:

```
int i = 42;
int &r = i;    // ok: r ссылается на i
int &&rr = i; // ошибка: нельзя связать ссылку на
               // r-значение
               // с l-значением
int &r2 = i * 42; // ошибка: i * 42 - это r-
                   // значение
const int &r3 = i * 42; // ok: ссылку на константу
                       // можно
                       // связать с r-значением
int &&rr2 = i * 42; // ok: связать rr2 с
                     // результатом умножения
```

Функции, возвращающие ссылки на l-значение, наряду с присвоением,

индексированием, обращением к значению, а также префиксные операторы инкремента и декремента являются примерами выражений, возвращающих l-значения. Ссылку на l-значение можно также связать с результатом любого из этих выражений.

Все функции, возвращающие не ссылочный тип, наряду с арифметическими, реляционными, побитовыми и постфиксными операторами инкремента и декремента возвращают r-значения. С этими выражениями нельзя связать ссылку на l-значение, но можно связать либо константную ссылку на l-значение, либо ссылку на r-значение.

l-значения — устойчивы; r-значения — эфемерны

Глядя на список выражений l- и r-значений, становится понятно, что l- и r-значения существенно отличаются друг от друга: у l-значений есть постоянное состояние, тогда как r-значения, литералы и временные объекты создаются лишь в ходе вычисления выражений.

Поскольку ссылки на r-значение могут быть связаны только с временным объектом, известно, что:

- упомянутый объект будет удален,
- у этого объекта не может быть других пользователей.

Совместно эти факты означают, что использующий ссылку на r-значение код способен получать ресурсы от объекта, на который ссылается ссылка.



Ссылки на r-значение ссылаются на объекты, которые будут вскоре удалены. Следовательно, можно "захватить" состояние объекта, связанного со ссылкой на r-значение.

Переменные являются l-значениями

Хотя мы редко думаем об этом, переменная — это выражение с одним операндом и без оператора. Подобно любому другому выражению, переменная как выражение имеет свойства l- и r-значения. Переменные как выражения — это l-значения. Удивительно, но как следствие невозможно связать ссылку на r-значение с переменной, определенной как тип ссылки на r-значение:

```
int &&rr1 = 42; // ok: литералы — это r-значения  
int &&rr2 = rr1; // ошибка: выражение rr1 — это l-
```

значение!

С учетом предыдущего наблюдения, согласно которому r-значения представляют эфемерные объекты, нет ничего удивительного в том, что переменная представляет собой l-значение. В конце концов, переменная сохраняется, пока не выйдет из области видимости.



Переменная — это l-значение; нельзя непосредственно связать ссылку на r-значение с переменной, *даже если эта переменная была определена как тип ссылки на r-значение*.



Библиотечная функция move()

Хотя нельзя непосредственно связать ссылку на r-значение с l-значением, можно явно привести l-значение к соответствующему типу ссылки на r-значение. Вызов новой библиотечной функции move(), определенной в заголовке `utility`, позволяет также получить ссылку на r-значение, привязанную к l-значению. Для возвращения ссылки на r-значение на данный объект функция move() использует средства, описываемые в разделе 16.2.6:

```
int &&rr3 = std::move(rr1); // ok
```

Вызов функции move() указывает компилятору, что имеющееся l-значение следует рассматривать как r-значение. Следует помнить, что приведенный выше вызов функции move() обещает не использовать rr1 ни для чего, кроме присвоения или удаления. После вызова функции move() нельзя сделать никаких предположений о значении уже перемещенного объекта.



Перемещенный объект можно удалить, а можно присвоить ему новое значение, но значение уже перемещенного объекта использовать нельзя.

Как уже упоминалось, для использования большинства имен из

библиотеки, включая функцию `move()` (см. раздел 13.5), не нужно предоставлять объявление `using` (см. раздел 3.1). Произойдет вызов функции `std::move()`, а не `move()`. Причины этого рассматриваются в разделе 18.2.3.



Код, применяющий функцию `move()`, должен использовать синтаксис `std::move()`, а не `move()`. Это позволит избежать возможных конфликтов имен.

Упражнения раздела 13.6.1

Упражнение 13.45. В чем разница между ссылкой на r-значение и ссылкой на l-значение.

Упражнение 13.46. Какой вид ссылки может быть связан со следующими инициализаторами?

```
int f();
vector<int> vi(100);
int? r1 = f();
int? r2 = vi[0];
int? r3 = r1;
int? r4 = vi[0] * f();
```

Упражнение 13.47. Снабдите конструктором копий и оператором присвоения копии класса `String` из упражнения 13.44 раздела 13.5, функции которого выводят сообщения при каждом вызове.

Упражнение 13.48. Определите вектор `vector<String>` и вызовите для него функцию `push_back()` несколько раз. Запустите программу и посмотрите, как часто копируются строки.



13.6.2. Конструктор перемещения и присваивание при перемещении

Подобно классу `string` (и другим библиотечным классам), наши собственные классы могут извлечь пользу из способности перемещения ресурсов вместо копирования. Чтобы позволить собственным типам операции перемещения, следует определить конструктор перемещения и оператор присваивания при перемещении. Эти члены подобны соответствующим функциям копирования, но они захватывают ресурсы заданного объекта, а не копируют их.



Как и у конструктора копий, у конструктора перемещения есть начальный параметр, являющийся ссылкой на тип класса. В отличие от конструктора копии, ссылочный параметр конструктора перемещения является ссылкой на `r`-значение. Подобно конструктору копий, у всех дополнительных параметров должны быть аргументы по умолчанию.

Кроме перемещения ресурсов, конструктор перемещения должен гарантировать такое состояние перемещенного объекта, при котором его удаление будет безопасно. В частности, сразу после перемещения ресурса оригинальный объект больше не должен указывать на перемещенный ресурс, ответственность за него принимает вновь созданный объект.

В качестве примера определим конструктор перемещения для класса `StrVec`, чтобы перемещать, а не копировать элементы из одного объекта класса `StrVec` в другой:

```
StrVec::StrVec( StrVec &&s) noexcept // перемещение
не будет передавать
                                                // исключений
// инициализаторы членов получают ресурсы из s
: elements(s.elements), first_free(s.first_free),
cap(s.cap) {
    // оставить s в состоянии, при котором запуск
деструктора безопасен
    s.elements = s.first_free = s.cap = nullptr;
}
```

Оператор `noexcept` (уведомляющий о том, что конструктор не передает исключений) описан ниже, а пока рассмотрим, что делает этот конструктор.

В отличие от конструктора копий, конструктор перемещения не резервирует новую память; он получает ее от заданного объекта класса `StrVec`. Получив область памяти от своего аргумента, тело конструктора присваивает указателям заданного объекта значение `nullptr`. После перемещения оригинальный объект продолжает существовать. В конечном счете оригинальный объект будет удален, а значит, будет выполнен его деструктор. Деструктор класса `StrVec` вызывает функцию `deallocate()` для указателя `first_free`. Если забыть изменить указатель `s.first_free`, то удаление оригинального объекта освободит область памяти, которая была только что передана.



Операции перемещения, библиотечные контейнеры и исключения

Поскольку операция перемещения выполняется при "захвате" ресурсов, она обычно не резервирует ресурсы. В результате операции перемещения обычно не передают исключений. Когда создается функция перемещения, неспособная передавать исключения, об этом факте следует сообщить библиотеке. Как будет описано вскоре, если библиотека не знает, что конструктор перемещения не будет передавать исключений, она предпримет дополнительные меры по отработке возможности передачи исключения при перемещении объекта этого класса.



Один из способов сообщить об этом библиотеке — определить оператор `noexcept` в конструкторе. Введенный новым стандартом оператор `noexcept` подробно рассматривается в разделе 18.1.4, а пока достаточно знать, что он позволяет уведомить, что функция не будет передавать исключений. Оператор `noexcept` указывают после списка параметров функции. В конструкторе его располагают между списком параметров и символом `:`, начинающим список инициализации конструктора:

```
class StrVec {  
public:  
    StrVec( StrVec&&)      noexcept;      // конструктор
```

```
пе ре ме ще ния  
    // другие члены, как  
пред же де  
} ;  
StrVec::StrVec( StrVec     &&s)      noexcept   :      /*  
инициализаторы членов */  
{ /* тело конструктора */ }
```

Оператор `noexcept` следует объявить и в заголовке класса, и в определении, если оно расположено вне класса.



Конструкторы перемещения и операторы присваивания при перемещении, которые не могут передавать исключения, должны быть отмечены как `noexcept`.

Понимание того, почему необходим оператор `noexcept`, может помочь углубить понимание того, как библиотека взаимодействует с объектами написанных вами типов. В основе требования указывать, что функция перемещения не будет передавать исключения, лежат два взаимосвязанных факта: во-первых, хотя функции перемещения обычно не передают исключений, им это разрешено. Во-вторых, библиотечные контейнеры предоставляют гарантии относительно того, что они будут делать в случае исключения. Например, класс `vector` гарантирует, что, если исключение произойдет при вызове функции `push_back()`, сам вектор останется неизменным.

Теперь рассмотрим происходящее в функции `push_back()`. Подобно соответствующей функции класса `StrVec` (см. раздел 13.5), функция `push_back()` класса `vector` могла бы потребовать пересоздания вектора. При пересоздании вектор перемещает элементы из прежней своей области памяти в новую, как в функции `reallocate()` (см. раздел 13.5).

Как только что упоминалось, перемещение объекта обычно изменяет состояние оригинального объекта. Если пересоздание использует конструктор перемещения и этот конструктор передает исключение после перемещения некоторых, но не всех элементов, возникает проблема. Перемещенные элементы в прежнем пространстве были бы изменены, а незаполненные элементы в новом пространстве еще не будут созданы. В

данном случае класс `vector` не удовлетворял бы требованию оставаться неизменным при исключении.

С другой стороны, если класс `vector` использует конструктор копий, то при исключении он может легко удовлетворить это требование. В данном случае, пока элементы создаются в новой памяти, прежние элементы остаются неизменными. Если происходит исключение, вектор может освободить зарезервированное пространство (оно могло бы и не быть успешно зарезервировано) и прекратить операцию. Элементы оригинального вектора все еще существуют.

Во избежание этой проблемы класс `vector` должен использовать во время пересоздания конструктор копий вместо конструктора перемещения, *если только не известно*, что конструктор перемещения типа элемента не может передать исключение. Если необходимо, чтобы объекты типа были перемещены, а не скопированы при таких обстоятельствах, как пересоздание вектора, то следует явно указать библиотеке, что использовать конструктор перемещения безопасно. Для этого конструктор перемещения (и оператора присваивания при перемещении) следует отметить как `noexcept`.

Оператор присваивания при перемещении

Оператор присваивания при перемещении делает то же, что и деструктор с конструктором перемещения. Подобно конструктору перемещения, если оператор присваивания при перемещении не будет передавать исключений, то его следует объявить как `noexcept`. Подобно оператору присвоения копии, оператор присваивания при перемещении должен принять меры против присвоения себя себе:

```
StrVec &StrVec::operator=(StrVec &&rhs) noexcept {
    // прямая проверка на присвоение себя себе
    if (this != &rhs) {
        free(); // освободить существующие элементы
        elements = rhs.elements; // получить ресурсы от
rhs
        first_free = rhs.first_free;
        cap = rhs.cap;
        // оставить rhs в удаляемом состоянии
        rhs.elements = rhs.first_free = rhs.cap =
nullptr;
    }
    return *this;
```

}

В данном случае осуществляется прямая проверка совпадения адресов в указателях `rhs` и `this`. Если это так, то правый и левый операнды относятся к тому же объекту, и делать ничего не надо. В противном случае следует освободить память, которую использовал левый операнд, а затем принять память от заданного объекта. Как и в конструкторе перемещения, указателю `rhs` присваивается значение `nullptr`.

Может показаться удивительным, что мы потрудились проверить присвоение себя самому. В конце концов, присваивание при перемещении требует для правого операнда `r`-значения. Проверка осуществляется потому, что то `r`-значение могло быть результатом вызова функции `move()`. Подобно любому другому оператору присвоения, крайне важно не освобождать ресурсы левого операнда прежде, чем использовать (возможно, те же) ресурсы правого операнда.



Исходный объект перемещения должен быть в удаляемом состоянии

Перемещение объекта не удаляет его оригинал: иногда после завершения операции перемещения оригинальный объект следует удалить. Поэтому, создавая функцию перемещения, следует гарантировать, что после перемещения оригинальный объект будет находиться в состоянии, допускающем запуск деструктора. Функция перемещения класса `StrVec` выполняет это требование и присваивает указателям-членам оригинального объекта значение `nullptr`.

Кроме гарантии безопасного удаления оригинального объекта, функции перемещения должны оставлять объект в допустимом состоянии. Обычно допустимым считается тот объект, которому может быть безопасно присвоено новое значение или который может быть использован другими способами, не зависящими от его текущего значения. С другой стороны, у функций перемещения нет никаких требований относительно значения, которое остается в оригинальном объекте. Таким образом, программы никогда не должны зависеть от значения оригинального объекта после перемещения.

Например, при перемещении объекта библиотечного класса `string` или контейнера известно, что оригинальный объект перемещения остается допустимым. В результате для оригинальных объектов перемещения можно выполнять такие функции, как `empty()` или `size()`. Однако

предсказать результат их выполнения затруднительно. Логично было бы ожидать, что оригинальный объект перемещения будет пуст, но это не гарантируется.

Функции перемещения класса `StrVec` оставляют оригинальный объект перемещения в том же состоянии, в котором он находился бы после инициализации по умолчанию. Поэтому все функции класса `StrVec` продолжат выполняться с его объектом точно так же, как с любым другим инициализированным по умолчанию объектом класса `StrVec`. Другие классы, с более сложной внутренней структурой, могут вести себя по-другому.



После операции перемещения "оригинальный объект" должен остаться корректным, допускающим удаление объектом, но для пользователей его значение непредсказуемо.

Синтезируемые функции перемещения

Подобно конструктору копий и оператору присвоения копии, компилятор способен сам синтезировать конструктор перемещения и оператор присваивания при перемещении. Однако условия, при которых он синтезирует функции перемещения, весьма отличаются от тех, при которых он синтезирует функции копирования.

Помните, что если не объявить собственный конструктор копий или оператор присвоения копии, компилятор *всегда* синтезирует их сам (см. раздел 13.1.1 и раздел 13.1.2). Функции копирования определяются или как функции почлененного копирования либо присвоения объекта, или как удаленные функции.

В отличие от функций копирования, для некоторых классов компилятор не синтезирует функции перемещения *вообще*. В частности, если класс определяет собственный конструктор копий, оператор присвоения копии или деструктор, конструктор перемещения и оператор присваивания при перемещении не синтезируются. В результате у некоторых классов нет конструктора перемещения или оператора присваивания при перемещении. Как будет продемонстрировано вскоре, когда у класса нет функции перемещения, вместо него в результате обычного подбора функции будет использована соответствующая функция копирования.

Компилятор синтезирует конструктор перемещения или оператор

присваивания при перемещении, *только если* класс не определяет ни одной из собственных функций-членов управления копированием и если каждая нестатическая переменная-член класса может быть перемещена. Компилятор может перемещать члены встроенного типа, а также члены типа класса, если у него есть соответствующая функция-член перемещения:

```
// компилятор синтезирует функции перемещения для X
и hasX
struct X {
    int i; // встроенные типы могут быть перемещены
    std::string s; // string определяет собственные
функции перемещения
};

struct hasX {
    X mem; // для X синтезированы функции перемещения
};

X x, x2 = std::move(x); // использует синтезируемый
конструктор
// перемещения
hasX hx, hx2 = std::move(hx); // использует
синтезируемый конструктор
// перемещения
```



ВНИМАНИЕ

Компилятор синтезирует конструктор перемещения и оператор присваивания при перемещении, только если класс не определяет ни одной из собственных функций-членов управления копированием и только если все переменные-члены могут быть созданы перемещением и присвоены при перемещении соответственно.

- В отличие от функций копирования, функции перемещения никогда не определяются неявно как удаленные. Но если явно запросить компилятор создать функцию перемещения, применив `= default` (см. раздел 7.1.4), но компилятор окажется неспособен переместить все члены, то функция перемещения будет определена как удаленная. Важное исключение из правила, согласно которому синтезируемая функция перемещения определяется как удаленная, подобно таковому для функций

копирования (см. раздел 13.1.6).

- В отличие от конструктора копий, конструктор перемещения определяется как удаленный, если у класса есть член, определяющий собственный конструктор копий, но не определяющий конструктор перемещения, или если у класса есть член, который не определяет собственные функции копирования и для которого компилятор неспособен синтезировать конструктор перемещения. То же относится к присваиванию при перемещении.

- Конструктор перемещения и оператор присваивания при перемещении определяются как удаленные, если у класса есть член, собственный конструктор перемещения которого или оператор присваивания при перемещении которого удален или недоступен.

- Как и конструктор копий, конструктор перемещения определяется как удаленный, если деструктор удален или недоступен.

- Как и оператор присвоения копии, оператор присваивания при перемещении определяется как удаленный, если у класса есть константный или ссылочный член.

Предположим, например, что в классе `Y` определен собственный конструктор копий, но не определен собственный конструктор перемещения:

```
// класс Y определяет собственный конструктор
// копий, но не конструктор
// перемещения
struct hasY {
    hasY() = default;
    hasY( hasY&&) = default;
    Y tem; // hasY будет иметь удаленный конструктор
    // перемещения
};

hasY hy, hy2 = std::move( hy); // ошибка:
// конструктор перемещения удален
```

Компилятор может скопировать объекты типа `Y`, но не может переместить их. Класс `hasY` явно запросил конструктор перемещения, который компилятор не способен создать. Следовательно, класс `hasY` получит удаленный конструктор перемещения. Если бы у класса `hasY` отсутствовало объявление конструктора перемещения, то компилятор не синтезировал бы конструктор перемещения вообще. Функции перемещения не синтезируются, если в противном случае они были

определенены как удаленные.

И последнее взаимоотношение между функциями перемещения и синтезируемыми функциями-членами управления копированием: тот факт, определяет ли класс собственные функции перемещения, влияет на то, как синтезируются функции копирования. Если класс определит любой конструктор перемещения и (или) оператор присваивания при перемещении, то синтезируемый конструктор копий и оператор присвоения копии для этого класса будут определены как удаленные.



Классы, определяющие конструктор перемещения или оператор присваивания при перемещении, должны также определять собственные функции копирования. В противном случае эти функции-члены по умолчанию удаляются.

R-значения перемещаются, а l-значения копируются...

Когда у класса есть и конструктор перемещения и конструктор копий, компилятор использует обычный подбор функции, чтобы выяснить, какой из конструкторов использовать (см. раздел 6.4). С присвоением точно так же. Например, в классе `StrVec` версия копирования получает ссылку на `const StrVec`. В результате она применима к любому типу, допускающему приведение к классу `StrVec`. Версия перемещения получает `StrVec&&` и применима только к аргументам r-значениям (неконстантным):

```
StrVec v1, v2;
v1 = v2;                                // v2 - l-значение;
присвоение копии
StrVec getVec(istream &); // getVec возвращает r-
значение
v2 = getVec(cin);                      // getVec(cin) - r-
значение;                                // присвоение перемещения
```

В первом случае оператору присвоения передается объект `v2`. Его типом является `StrVec`, а выражение `v2` является l-значением. Версия присвоения при перемещении не является подходящей (см. раздел 6.6), поскольку нельзя неявно связать ссылку на r-значение с l-значением.

Следовательно, в этом случае используется оператор присвоения копии.

Во втором случае присваивается результат вызова функции `getVec()`, — это *r*-значение. Теперь подходящими являются оба оператора присвоения — результат вызова функции `getVec()` можно связать с любым параметром оператора. Вызов оператора присвоения копии требует преобразования в константу, в то время как `StrVec&&` обеспечивает точное соответствие. Следовательно, второе присвоение использует оператор присваивания при перемещении.

***...но r-значения копируются, если нет конструктора
перемещения***

Что если класс имеет конструктор копий, но не определяет конструктор перемещения? В данном случае компилятор не будет синтезировать конструктор перемещения. Это значит, что у класса есть конструктор копий, но нет конструктора перемещения. Если у класса нет конструктора перемещения, подбор функции гарантирует, что объекты этого типа будут копироваться, даже при попытке перемещения их вызовом функции `move()`:

```
class Foo {  
public:  
    Foo() = default;  
    Foo( const Foo& ); // конструктор копий  
    // другие члены, но Foo не определяет конструктор  
    // перемещения  
};  
Foo x;  
Foo y( x ); // конструктор копий; x - это  
// 1-значение  
Foo z( std::move( x ) ); // конструктор копий,  
// поскольку конструктора  
// перемещения нет
```

Вызов функции `move(x)` при инициализации объекта `z` возвращает указатель `FOO&&`, привязанный к объекту `x`. Конструктор копий для класса `Foo` является подходящим, поскольку вполне допустимо преобразовать `FOO&&` в `const Foo&`. Таким образом, инициализация объекта `z` использует конструктор копий класса `Foo`.

Следует заметить, что использование конструктора копий вместо конструктора перемещения почти безусловно безопасно (то же

справедливо и для оператора присвоения). Обычно конструктор копий отвечает требованиям соответствующего конструктора перемещения: он копирует заданный объект и оставляет оригинальный объект в допустимом состоянии. Конструктор копий, напротив, не будет изменять значение оригинального объекта.



Если у класса будет пригодный конструктор копий и не будет конструктора перемещения, то объекты будут перемещены конструктором копий. То же справедливо для оператора присвоения копии и присвоения при перемещении.

Операторы присвоения копии и обмена и перемещение

Версия класса `HasPtr`, определявшая *оператор присвоения копии и обмена* (copy-and-swap assignment operator) (см. раздел 13.3), — хорошая иллюстрация взаимодействия механизма подбора функции и функций перемещения. Если в этот класс добавить конструктор перемещения, то фактически будет получен также оператор присваивания при перемещении:

```
class HasPtr {
public:
    // добавлен конструктор перемещения
    HasPtr(HasPtr &&p) noexcept : ps(p.ps), i(p.i)
{ p.ps = 0; }
    // оператор присвоения - и оператор перемещения, и
    // присвоения копии
    HasPtr& operator=(HasPtr rhs)
    { swap(*this, rhs); return *this; }
    // другие члены как в р. 13.2.1
};
```

В этой версии класса добавлен конструктор перемещения, получающий значения из своего аргумента. Тело конструктора обнуляет указатель-член данного объекта класса `HasPtr`, чтобы гарантировать безопасное удаление оригинального объекта перемещения. Эта функция не делает ничего, она не может передать исключение, поэтому отметим ее как `noexcept` (см. раздел 13.6.2).

Теперь рассмотрим оператор присвоения. У него есть не ссылочный параметр, а значит, этот параметр инициализируется копией (см. раздел 13.1.1). В зависимости от типа аргумента инициализация копией использует либо конструктор копий, либо конструктор перемещения; l-значения копируются, а r-значения перемещаются. В результате этот оператор однократного присвоения действует и как присвоение копии, и как присваивание при перемещении.

Предположим, например, что объекты `hp` и `hp2` являются объектами класса `HasPtr`:

```
hp = hp2; // hp2 - l-значение; для копирования hp2  
используется
```

// конструктор копий

```
hp = std::move( hp2 ); // hp2 перемещает конструктор  
перемещения
```

В первом случае присвоения правый операнд — l-значение, поэтому конструктор перемещения не подходит. Для инициализации `rhs` будет использоваться конструктор копий. Он будет резервировать новую строку и копировать ту строку, на которую указывает `hp2`.

Во втором случае присвоения вызывается функция `std::move()` для связывания ссылки на r-значение с объектом `hp2`. В данном случае подходят и конструктор копий, и конструктор перемещения. Но поскольку аргумент — это ссылка на r-значение, точное соответствие обеспечит конструктор перемещения. Конструктор перемещения копирует указатель из объекта `hp2` и не резервирует память.

Независимо от того, использовался ли конструктор копии или перемещения, тело оператора присвоения обменивает содержимое двух своих operandов. Обмен объектов класса `HasPtr` приводит к обмену указателями-членами и переменными-членами (типа `int`) этих двух объектов. После вызова функции `swap()` правый operand будет содержать указатель на строку, который ранее принадлежал левому. При выходе `rhs` из области видимости эта строка будет удалена.

Совет. Обновленное правило трех

Все пять функций-членов управления копированием можно считать единым блоком: если класс определяет любую из этих функций, он должен обычно определять их все. Как уже упоминалось, для правильной работы некоторые классы *должны* определять конструктор копий, оператор присвоения копии и деструктор (см. раздел 13.6). Как

правило, у таких классов есть ресурс, который должны копировать функции-члены копирования. Обычно копирование ресурса влечет за собой некоторые дополнительные затраты. Классы, определяющие конструктор перемещения и оператор присваивания при перемещении, могут избежать этих затрат в тех обстоятельствах, где копия не обязательна.

Функции перемещения для класса Message

Классы, определяющие собственный конструктор копий и оператор присвоения копии, обычно определяют и функции перемещения. Например, наши классы `Message` и `Folder` (см. раздел 13.4), должны определять функции перемещения. При определении функций перемещения класс `Message` может использовать функции перемещения классов `string` и `set`, чтобы избежать дополнительных затрат при копировании членов `contents` и `folders`.

Но в дополнение к перемещению члена `folders` следует также обновить каждый объект класса `Folder`, указывавший на оригинал объекта класса `Message`. Следует также удалить указатели на прежний объект класса `Message` и добавить указатели на новый.

И конструктор перемещения, и оператор присваивания при перемещении должны обновлять указатели `Folder`, поэтому начнем с определения функций для выполнения этих действий:

```
// переместить указатели Folder из m в данное
Message
void Message::move_Folders( Message *m) {
    folders = std::move( m->folders); // использует
присвоение перемещения
                                                // класса set
    for ( auto f : folders) { // для каждого Folder
        f->remMsg( m);      // удалить старый Message из
Folder
        f->addMsg( this); // добавить этот Message в этот
Folder
    }
    m->folders.clear(); // гарантировать безопасное
удаление m
}
```

Функция начинает работу с перемещения набора `folders`. При вызове

функции `move()` используется оператор присвоения при перемещении класса `set`, а не его оператор присвоения копии. Если пропустить вызов функции `move()`, код все равно будет работать, но осуществляя ненужное копирование. Затем функция перебирает папки, удаляя указатель на оригинал сообщения и добавляя указатель на новое сообщение.

Следует заметить, что вставка элемента в набор может привести к передаче исключения, поскольку добавление элемента на контейнер требует резервирования памяти, вполне может быть передано исключение `bad_alloc` (см. раздел 12.1.2). Таким образом, в отличие от функций перемещения классов `HasPtr` и `StrVec`, конструктор перемещения и операторы присваивания при перемещении класса `Message` могли бы передать исключения, поэтому не будем отмечать их как `noexcept` (см. раздел 13.6.2).

Функция заканчивается вызовом функции `clear()` объекта `m.folders`. Известно, что после перемещения объект `m.folders` вполне допустим, но его содержимое непредсказуемо. Поскольку деструктор класса `Message` перебирает набор `folders`, необходимо убедиться, что набор пуст.

Конструктор перемещения класса `Message` вызывает функцию `move()`, чтобы переместить содержимое и инициализировать по умолчанию свой член `folders`:

```
Message::Message( Message && m) :  
contents( std::move( m.contents)) {  
    move_Folders( &m); // переместить folders и  
обновить указатели Folder  
}
```

В теле конструктора происходит вызов функции `move_Folders()`, чтобы удалить указатели на `m` и вставить указатели на данное сообщение.

Оператор присваивания при перемещении непосредственно проверяет случай присвоения себя себе:

```
Messages Message::operator=( Message && rhs) {  
    if (this != &rhs) { // прямая проверка присвоения  
себя себе  
        remove_from_Folders();  
        contents = std::move( rhs.contents); // присвоение  
при перемещении  
        move_Folders( &rhs); // сбросить папки, чтобы  
указывать на это
```

```
// сообщение
}
return *this;
}
```

Подобно любым операторам присвоения, оператор присваивания при перемещении должен удалить прежние данные левого операнда. В данном случае удаление левого операнда требует удаления указателей на это сообщение из существующих папок, что и делает вызов функции `remove_from_Folders()`. После удаления из папок происходит вызов функции `move()`, чтобы переместить `contents` из объекта `rhs` в `this`. Остается только вызвать функцию `move_Folders()`, чтобы модифицировать указатели `Folder`.

Итераторы перемещения

Функция `reallocate()` класса `StrVec` (см. раздел 13.5) использовала вызов функции `construct()` в цикле `for` для копирования элементов из прежней памяти в новую. Альтернативой циклу был бы просто вызов функции `uninitialized_copy()` для создания нового пространства в памяти. Однако функция `uninitialized_copy()` делает именно то, о чём говорит ее имя: она копирует элементы. Нет никакой аналогичной библиотечной функции для перемещения объектов в пустую память.



Вместо нее новая библиотека определяет адаптер *итератора перемещения* (`move iterator`) (см. раздел 10.4). Итератор перемещения адаптирует переданный ему итератор, изменяя поведение его оператора обращения к значению. Обычно оператор обращения к значению итератора возвращает ссылку на l-значение элемента. В отличие от других итераторов, оператор обращения к значению итератора перемещения возвращает ссылку на r-значение.

Обычный итератор преобразуется в итератор перемещения при вызове библиотечной функции `make_move_iterator()`, которая получает итератор и возвращает итератор перемещения.

Все остальные функции первоначального итератора работают, как обычно. Поскольку эти итераторы поддерживают обычные функции итераторов, пару итераторов перемещения вполне можно передать алгоритму. В частности, итераторы перемещения можно передать

```

алгоритму uninitialized_copy( ) :
    void StrVec::reallocate() {
        // зарезервировать вдвое больше пространства, чем
для текущего
        // количества элементов
        auto newcapacity = size() ? 2 * size() : 1;
        auto first = alloc.allocate( newcapacity );
        // переместить элементы
        auto last = free( ); // освободить прежнее
пространство
        elements = first; // обновить указатели
        first_free = last;
        cap = elements + newcapacity;
    }

```

Алгоритм `uninitialized_copy()` вызывает функцию `construct()` для каждого элемента исходной последовательности, чтобы скопировать элемент по назначению. Для выбора элемента из исходной последовательности данный алгоритм использует оператор обращения к значению итератора. Поскольку был передан итератор перемещения, оператор обращения к значению возвращает ссылку на г-значение. Это означает, что функция `construct()` будет использовать для создания элементов конструктор перемещения.

Следует заметить, что стандартная библиотека не дает гарантий применимости всех алгоритмов с итераторами перемещения. Так как перемещение объекта способно удалить оригинал, итераторы перемещения следует передать алгоритмам, только тогда, когда вы уверены, что алгоритм не будет обращаться к элементам после того, как он присвоил этот элемент или передал его пользовательской функции.

Совет. Не слишком спешите с перемещением

Поскольку состояние оригинального объекта перемещения неопределенно, вызов для него функции `std::move()` — опасная операция. Когда происходит вызов функции `move()`, следует быть абсолютно уверенными в том, что у оригинального объекта перемещения не может быть никаких других пользователей.

Взвешенно использованная в коде класса, функция `move()` способна обеспечить существенный выигрыш в производительности. Небрежное ее использование в обычном пользовательском коде (в отличие от кода реализации класса), вероятней всего, приведет к загадочным и трудно обнаруживаемым ошибкам, а не к повышению производительности приложения.



Рекомендуем

За пределами кода реализации класса, такого как конструкторы перемещения или операторы присваивания при перемещении, используйте функцию `std::move()` только при абсолютной уверенности в необходимости перемещения и в том, что перемещение гарантированно будет безопасным.

Упражнения раздела 13.6.2

Упражнение 13.49. Добавьте конструктор перемещения и оператор присваивания при перемещении в классы `StrVec`, `String` и `Message`.

Упражнение 13.50. Снабдите функции перемещения класса `String` операторами вывода и снова запустите программу из упражнения 13.48 раздела 13.6.1, в котором использовался вектор `vector<String>`, и посмотрите, когда теперь удается избежать копирования.

Упражнение 13.51. Хотя указатель `unique_ptr` не может быть скопирован, в разделе 12.1.5 была написана функция `clone()`, которая возвратила указатель `unique_ptr` по значению. Объясните, почему эта функция допустима и как она работает.

Упражнение 13.52. Объясните подробно, что происходит при присвоении объектов класса `HasPtr`. В частности, опишите шаг за шагом, что происходит со значениями `hp`, `hp2` и параметром `rhs` в операторе присвоения класса `HasPtr`.

Упражнение 13.53. С точки зрения низкоуровневой эффективности оператор присвоения класса `HasPtr` не идеален. Объясните почему. Реализуйте для класса `HasPtr` оператор присвоения копии и присваивания при перемещении и сравните действия, выполняемые в новом операторе присваивания при перемещении, с версией копии и обмена.

Упражнение 13.54. Что бы случилось, если бы мы определи оператор присваивания при перемещении для класса `HasPtr`, но не изменили

оператор копии и обмена? Напишите код для проверки вашего ответа.



13.6.3. Ссылки на r-значение и функции-члены

Все функции-члены, кроме конструкторов и операторов присвоения, могут извлечь пользу из предоставления версии копирования и перемещения. Такие функции-члены с поддержкой перемещения обычно используют ту же схему параметров, что и конструктор копий/перемещения и операторы присвоения, — одна версия получает ссылку на константное l-значение, а вторая — ссылку на не константное r-значение.

Например, библиотечные контейнеры, определяющие функцию `push_back()`, предоставляют две версии: параметр одной является ссылкой на r-значение, а другой — ссылкой на константное l-значение. С учетом того, что `X` является типом элемента, эти функции контейнера определяются так:

```
void push_back( const X& ) ; // копирование: привязка  
к любому X  
void push_back( X&& ) ; // перемещение: привязка  
только к изменяемым  
// r-значениям типа X
```

Первой версии функции `push_back()` можно передать любой объект, который может быть приведен к типу `X`. Эта версия копирует данные своего параметра. Второй версии можно передать только r-значение, которое не является константой. Эта версия точнее и лучшее соответствует неконстантным r-значениям и будет выполнена при передаче поддающегося изменению r-значения (см. раздел 13.6.2). Эта версия способна захватить ресурсы своего параметра.

Обычно нет никакой необходимости определять версии функций получающих `const X&&` или просто `X&&`. Обычно ссылку на r-значение передают при необходимости "захватить" аргумент. Для этого аргумент не должен быть константой. Точно так же копирование объекта не должно изменять скопированный объект. В результате обычно нет никакой необходимости определять версию, получающую простой параметр `X&`.



У перегруженных функций, различающих перемещение и копирование

параметра, обычно есть одна версия, получающая параметр типа `const T&`, и вторая, получающая параметр типа `T&&`.

В качестве более конкретного примера придадим классу `StrVec` вторую версию функции `push_back()`:

```
class StrVec {
public:
    void push_back(const std::string&); // копирует
элемент
    void push_back(std::string&&); // перемещает
элемент
    // другие члены как прежде
};

// неизменно с оригинальной версией в разделе 13.5
void StrVec::push_back(const string& s) {
    chk_n_alloc(); // удостовериться в наличии места
для другого элемента
    // создать копию s в элементе, на который
указывает first_free
    alloc.construct(first_free++, s);
}
void StrVec::push_back(string &&s) {
    chk_n_alloc(); // пересоздает StrVec при
необходимости
    alloc.construct(first_free++, std::move(s));
}
```

Эти функции-члены почти идентичны. Различие в том, что версия ссылки на `r`-значение функции `push_back()` вызывает функцию `move()`, чтобы передать этот параметр функции `construct()`. Как уже упоминалось, функция `construct()` использует тип своего второго и последующих аргументов для определения используемого конструктора. Поскольку функция `move()` возвращает ссылку на `r`-значение, аргумент функции `construct()` будет иметь тип `string&&`. Поэтому для создания нового последнего элемента будет использован конструктор перемещения класса `string`.

Когда вызывается функция `push_back()`, тип аргумента определяет, копируется ли новый элемент в контейнер или перемещается:

```
StrVec vec; // пустой StrVec
```

```

string s = "some string or another";
vec.push_back(s); // вызов push_back(const string&)
vec.push_back("done"); // вызов push_back(string&&)

```

Эти вызовы различаются тем, является ли аргумент l-значением (`s`) или r-значением (временная строка, созданная из слова "done"). Вызовы распознаются соответственно.

Ссылки на l-значения, r-значения и функции-члены

Обычно функцию-член объекта можно вызвать независимо от того, является ли этот объект l- или r-значением. Например:

```

string s1 = "a value", s2 = "another";
auto n = (s1 + s2).find('a');

```

Здесь происходит вызов функции-члена `find()` (см. раздел 9.5.3) для r-значения класса `string`, полученного при конкатенации двух строк. Иногда такой способ применения может удивить:

```
s1 + s2 = "wow!";
```

Здесь r-значению присваивается результат конкатенации двух строк.

До нового стандарта не было никакого способа предотвратить подобное применение. Для обеспечения совместимости с прежней версией библиотечные классы продолжают поддерживать присвоение r-значению; в собственных классах такое может понадобиться предотвратить. В таком случае левый операнд (т.е. объект, на который указывает указатель `this`) обязан быть l-значением.



Свойство l- или r-значения указателя `this` задают таким же образом, как и константность функции-члена (см. раздел 7.1.2): помещая *квалификатор ссылки* (reference qualifier) после списка параметров:

```

class Foo {
public:
    Foo &operator=(const Foo&) &; // возможно
присвоение только
// изменяемым l-
значениям
// другие члены класса Foo
};

Foo &Foo::operator=(const Foo &rhs) & {

```

```

    // сделать все необходимое для присвоения rhs
этому объекту
    return *this;
}

```

Квалификаторы ссылки & или && означают, что указатель `this` может указывать на l- или r-значение соответственно. Подобно спецификатору `const`, квалификатор ссылки может быть применен только к (нестатической) функции-члену и должен присутствовать как в объявлении, так и в определении функции.

Функцию, квалифицированную символом &, можно применить только к l-значению, а функцию, квалифицированную символом &&, — только к r-значению:

```

Foo &retFoo(); // возвращает ссылку;
                // вызов retFoo() является l-
значением
Foo retVal(); // возвращает значение; вызов
retVal() - r-значение
Foo i, j; // i и j - это l-значения
i = j; // ok: i - это l-значение
retFoo() = j; // ok: retFoo() возвращает l-
значение
retVal() = j; // ошибка: retVal() возвращает r-
значение
i = retVal(); // ok: вполне можно передать r-
значение как правый
                // операнд присвоения

```

Функция может быть квалифицирована и ссылкой, и константой. В таких случаях квалификатор ссылки должен следовать за спецификатором `const`:

```

class Foo {
public:
    Foo someMem() & const; // ошибка: первым должен
быть
                                // спецификатор const
    Foo anotherMem() const &; // ok: спецификатор
const расположен первым
} ;

```

Перегрузка и ссылочные функции

Подобно тому, как можно перегрузить функцию-член на основании константности параметра (см. раздел 7.3.2), ее можно перегрузить на основании квалификатора ссылки. Кроме того, функцию можно перегрузить на основании квалификатора ссылки и константности. В качестве примера придадим классу `Foo` член типа `vector` и функцию `sorted()`, возвращающую копию объекта класса `Foo`, в котором сортируется вектор:

```
class Foo {
public:
    Foo sorted() &&;           // применимо к изменяемым r-
значенням
    Foo sorted() const &;      // применимо к любому
объекту класса Foo
    // другие члены класса Foo
private:
    vector<int> data;
};

// этот объект - r-значение, поэтому его можно
сортировать на месте
Foo Foo::sorted() && {
    sort(data.begin(), data.end());
    return *this;
}
// этот объект либо константа, либо l-значение;
// так или иначе, его нельзя сортировать на месте
Foo Foo::sorted() const & {
    Foo ret(*this);           // создает
копию
    sort(ret.data.begin(), ret.data.end()); // //
сортирует копию
    return ret;                // //
возвращает копию
}
```

При выполнении функции `sorted()` для `r`-значения вполне безопасно сортировать вектор-член `data` непосредственно. Объект является `r`-значением, а это означает, что у него нет никаких других пользователей, поэтому данный объект можно изменить непосредственно. При выполнении функции `sorted()` для константного `r`- или `l`-значения

изменить этот объект нельзя, поэтому перед сортировкой вектор-член `data` необходимо скопировать.

Поиск перегруженной функции использует свойство l-значение/r-значение объекта, вызвавшего функцию `sorted()` для определения используемой версии:

```
retVal().sorted(); // retVal() - это r-value, вызов
Foo::sorted() &&
retFoo().sorted(); // retFoo() - это l-value,
// вызов Foo::sorted() const &
```

При определении константных функций-членов можно определить две версии, отличающиеся только тем, что одна имеет квалификатор `const`, а другая нет. Для ссылочной квалификации функций ничего подобного по умолчанию нет. При определении двух или более функций-членов с тем же именем и тем же списком параметров следует предоставить квалификатор ссылки для всех или ни для одной из этих функций:

```
class Foo {
public:
    Foo sorted() &&;
    Foo sorted() const; // ошибка: должен быть
квалификатор ссылки
    // Comp - псевдоним для типа функции (см. р. 6.7)
    // он применим для сравнения целочисленных
значений
    using Comp = bool(const int&, const int&);
    Foo sorted(Comp*); // ok: другой список
параметров
    Foo sorted(Comp*) const; // ok: ни одна из версий
не квалифицирована
                                            // как ссылка
};
```

Здесь объявление константной версии функции `sorted()` без параметров является ошибкой. Есть вторая версия функции `sorted()` без параметров, и у нее есть квалификатор ссылки, поэтому у константной версии этой функции также должен быть квалификатор ссылки. С другой стороны, те версии функции `sorted()`, которые получают указатель на функцию сравнения, прекрасно работают, поскольку ни у одной из функций нет спецификатора.



Если у функции-члена есть квалификатор ссылки, то у всех версий этой функции-члена с тем же списком параметров должны быть квалификаторы ссылки.

Упражнения раздела 13.6.3

Упражнение 13.55. Добавьте в класс StrBlob функцию `push_back()` в версии ссылки на `g`-значение.

Упражнение 13.56. Что бы было при таком определении функции `sorted()`:

```
Foo Foo::sorted() const & {
    Foo ret(*this);
    return ret.sorted();
}
```

Упражнение 13.57. Что если бы функция `sorted()` была определена так:

```
Foo Foo::sorted() const & {
    return Foo(*this).sorted();
}
```

Упражнение 13.58. Напишите версию класса `Foo` с операторами вывода в функциях `sorted()`, чтобы проверить свои ответы на два предыдущих упражнения.

Резюме

Каждый класс контролирует происходящее при копировании, перемещении, присвоении и удалении объектов его типа. Эти действия определяют специальные функции-члены: конструктор копий, конструктор перемещения, оператор присвоения копии, оператор присваивания при перемещении и деструктор. Конструктор перемещения и оператор присваивания при перемещении (обычно неконстантный) получают ссылку на `g`-значение; версии оператора копирования (обычно константные) получают обычную ссылку на `l`-значение.

Если класс не объявляет ни одну из этих функций, то компилятор определит их автоматически. Если они не определены как удаленные, эти

функции-члены инициализирует, перемещают, присваивают и удаляют объект, обрабатывая каждую нестатическую переменную-член по очереди. Синтезируемая функция делает то, что соответствует типу элемента для перемещения, копирования, присвоения и удаления этого элемента.

Классы, резервирующие память или другие ресурсы, почти всегда требуют, чтобы класс определил функции-члены управления копированием для управления зарезервированным ресурсом. Если класс нуждается в деструкторе, то он почти наверняка должен определить конструкторы перемещения и копирования, а также операторы перемещения и присвоения копии.

Термины

Деструктор (destructor). Специальная функция-член, освобождающая занятую объектом память, когда он выходит из области видимости или удаляется. Компилятор автоматически удаляет каждый член класса. При удалении переменных-членов типа класса используются их собственные деструкторы, а при удалении переменных-членов встроенного или составного типа конструктор ничего не делает. В частности, объект, на который указывает указатель-член класса, автоматически не удаляется деструктором.

Инициализация копией (copy initialization). Форма инициализации с использованием оператора = и предоставления инициализатора для создаваемого объекта. Используется также при передаче и возвращении объекта по значению, при инициализации массива или агрегатного класса. Инициализация копией использует конструктор копий или конструктор перемещения, в зависимости от того, является ли инициализатор l- или r-значением.

Итератор перемещения (move iterator). Адаптер, позволяющий создать итератор, обращение к значению которого возвращает ссылку на r-значение.

Квалификатор ссылки (reference qualifier). Символ, обычно указывающий, что нестатическая функция-член может быть вызвана для l- или r-значения. Спецификатор & или && следует за списком параметров или спецификатором const, если он есть. Функция с квалификатором & может быть вызвана только для l-значений, а функция с квалификатором && — только для r-значений.

Конструктор копий (copy constructor). Конструктор, который инициализирует новый объект как копию другого объекта того же типа.

При передаче объекта в функцию или из функции конструктор копий применяется неявно. Если конструктор копий не определен явно, компилятор синтезирует его самостоятельно.

Конструктор перемещения (*move constructor*). Конструктор, получающий ссылку на *г*-значение своего типа. Как правило, конструктор перемещения перемещает данные своего параметра во вновь созданный объект. После перемещения запуск деструктора для правого операнда должен быть безопасен.

Копирование и обмен (*copy and swap*). Техника написания операторов присвоения за счет копирования правого операнда, сопровождаемого вызовом функции *swap()*, обменивающей копию с левым операндом.

Оператор присваивания при перемещении (*move-assignment operator*). Версия оператора присвоения, получающая ссылку *г*-значения на ее тип. Как правило, оператор присваивания при перемещении перемещает данные из правого операнда в левый. После присвоения запуск деструктора для правого операнда должен быть безопасен.

Оператор присвоения копии (*copy-assignment operator*). Версия оператора присвоения, получающая объект того же типа, что и у нее. Обычно оператор присвоения копии имеет параметр, являющийся ссылкой на константу, и возвращает ссылку на свой объект. Компилятор сам синтезирует оператор присвоения копии, если класс не предоставляет его явно.

Перегруженный оператор (*overloaded operator*). Функция, переопределяющая один из операторов для работы с операндами данного класса. В этой главе описано определение лишь оператора присвоения, а более подробно перегрузка операторов рассматривается в главе 14.

Почленное копирование и присвоение (*memberwise copy/assign*). Так работают синтезируемые конструкторы копирования и перемещения, а также операторы присваивания при перемещении и копии. Перебирая все нестатические переменные-члены по очереди, синтезируемый конструктор копий или перемещения инициализирует каждую из них, копируя или при перемещении соответствующее значение из заданного объекта; оператор присваивания при перемещении и копии присваивают при перемещении или копируют каждую переменную-член правого объекта в левый. Инициализация и присвоение переменных-членов встроенного или составного типа осуществляются непосредственно, а членов типа класса — с использованием соответствующего конструктора перемещения или копирования либо оператора присвоения копии или присваивания при перемещении.

Синтезируемые конструкторы копирования и перемещения (synthesized copy/move constructor). Версии конструкторов копирования и перемещения, синтезируемые компилятором для классов, которые не определяют соответствующие конструкторы явно. Если они не определены как удаленные функции, синтезируемые конструкторы копирования и перемещения почленно инициализируют новый объект, копируя или перемещая члены из заданного объекта.

Синтезируемый деструктор (synthesized destructor). Версия деструктора, создаваемая (синтезируемая) компилятором для классов, в которых он не определен явно. Тело синтезируемого деструктора пусто.

Синтезируемый оператор присвоения (synthesized assignment operator). Версия оператора присвоения, создаваемого (синтезируемого) компилятором для классов, у которых он не определен явно. Если он не определен как удаленная функция, синтезируемый оператор присвоения почленно присваивает (перемещает) правый operand левому.

Ссылка на l-значение (l-value reference). Ссылка, которая может быть связана с l-значением.

Ссылка на r-значение (r-value reference). Ссылка на объект, который будет удален.

Счетчик ссылок (reference count). Программное средство, обычно используемое в членах управления копированием. Счетчик ссылок отслеживает количество объектов, совместно использующих некую сущность. Конструкторы (кроме конструкторов копирования и перемещения) устанавливают счетчик ссылок в 1. Каждый раз, когда создается новая копия, значение счетчика увеличивается. Когда объект удаляется, значение счетчика уменьшается. Оператор присвоения и деструктор проверяют, не достиг ли декремент счетчика ссылок нуля, и если это так, то они удаляют объект.

Удаленная функция (deleted function). Функция, которая не может быть использована. Для удаления функции в ее объявление включают часть = `delete`. Обычно удаленные функции используют для запрета компилятору синтезировать операторы копирования и (или) перемещения для класса.

Управление копированием (copy control). Специальные функции-члены, которые определяют действия, осуществляемые при копировании, присвоении и удалении объектов класса. Если эти функции не определены в классе явно, компилятор синтезирует их самостоятельно.

Функция move(). Библиотечная функция, обычно используемая для связи ссылки r-значения с l-значением. Вызов функции `move()` неявно

обещает, что объект не будет использован для перемещения, кроме его удаления или присвоения нового значения.

Глава 14

Перегрузка операторов и преобразований

Как упоминалось в главе 4, язык C++ предоставляет для встроенных типов множество операторов и автоматических преобразований. Они позволяют создавать разнообразные выражения, где используются разные типы данных.

Язык C++ позволяет переопределять смысл операторов, применяемых для объектов типа класса, а также определять для класса функции преобразования типов. Функции преобразования типа класса используются подобно встроенным преобразованиям для неявного преобразования (при необходимости) объекта одного типа в другой.

Перегрузка оператора (overloaded operator) позволяет определить смысл оператора, когда он применяется к операнду (операндам) типа класса. Разумное применение перегрузки операторов способно упростить программы, облегчить их написание и чтение. Например, поскольку наш первоначальный класс `Sales_item` (см. раздел 1.5.1) определял операторы ввода, вывода и суммы, сумму двух объектов класса `Sales_item` можно вывести так:

```
cout << item1 + item2; // вывод суммы двух объектов  
класса Sales_item
```

Класс `Sales_data` (см. раздел 7.1), напротив, еще не имеет перегруженных операторов, поэтому код вывода суммы его объектов окажется более подробным, а следовательно, менее ясным:

```
print( cout, add( data1, data2 ) ); // вывод суммы двух  
объектов
```

```
// класса
```

```
Sales_data
```



14.1. Фундаментальные концепции

Перегруженный оператор — это функция со специальным именем, состоящим из ключевого слова `operator`, сопровождаемого символом определяемого оператора. Подобно любой другой функции, перегруженный оператор имеет тип возвращаемого значения и список параметров.

Количество параметров функции перегруженного оператора совпадает с количеством operandов оператора. У унарного оператора — один параметр; у бинарного — два. В бинарном операторе левый operand передается первому параметру, а правый operand — второму. За исключением перегруженного оператора вызова функции, `operator()`, у перегруженного оператора не может быть аргументов по умолчанию (см. раздел 6.5.1).

Если перегруженный оператор является функцией-членом, то первый (левый) operand связывается с неявным указателем `this` (см. раздел 7.1.2). Поскольку первый operand неявно связан с указателем `this`, функция оператора-члена будет иметь на один явный параметр меньше, чем operandов у оператора.



Когда перегруженный оператор является функцией-членом, указатель `this` соответствует левому operandу. У операторов-членов на один явный параметр меньше, чем operandов.

Функция оператора должна быть либо членом класса, либо иметь по крайней мере один параметр типа класса:

// ошибка: переопределить встроенный оператор для целых чисел

```
int operator*(int, int);
```

Это ограничение означает невозможность изменить смысл оператора, относящегося к operandам встроенного типа.

Перегрузить можно многие, но не все операторы. Табл. 14.1

демонстрирует, может ли оператор быть перегружен. Перегрузка операторов `new` и `delete` рассматривается в разделе 19.1.1.

Перегрузить можно только существующие операторы и нельзя изобрести новые символы операторов. Например, нельзя определить оператор `operator**` для возведения числа в степень.

Таблица 14.1. Операторы

Операторы, которые могут быть перегружены					
+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=
=	* =	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []
Операторы, которые не могут быть перегружены					
::	. *	.		?:	

Четыре символа (`+`, `-`, `*` и `&`) служат и унарными операторами, и бинарными. Перегружен может быть один или оба из этих операторов. Определяемый оператор задает количество параметров:

`x == y + z;`

Это будет эквивалентно `x == (y + z)`.

Непосредственный вызов функции перегруженного оператора

Обычно функцию перегруженного оператора вызывают косвенно, применив оператор к аргументам соответствующего типа. Но функцию перегруженного оператора можно также вызвать непосредственно, как обычную функцию. Достаточно указать имя функции и передать соответствующее количество аргументов соответствующего типа:

```
// эквивалент вызова функции оператора, не
являющегося членом класса
data1 + data2; // обычное выражение
operator+(data1, data2); // эквивалентный вызов
функции
```

Эти вызовы эквивалентны: оба они являются вызовом функции `operator+()` с передачей `data1` как первого аргумента и `data2`, так и второго.

Явный вызов функции оператора-члена осуществляется таким же образом, как и вызов любой другой функции-члена: имя объекта (или указателя), для которого выполняется функция, и оператор точки (или стрелки) для выбора функции, которую следует вызвать:

```
data1 += data2;           // вызов на базе выражения  
data1.operator+=( data2 ); // эквивалентный вызов  
функции оператора-члена
```

Каждый из этих операторов вызывает функцию-член `operator+=`, где указатель `this` содержит адрес объекта `data1`, а объект `data2` передан как аргумент.

Некоторые операторы не следуют перегружать

Помните, что некоторые операторы гарантируют порядок вычисления operandов. Поскольку использование перегруженного оператора на самом деле является вызовом функции, эти гарантии не распространяются на перегруженные операторы. В частности, гарантии вычисления operandов логических операторов AND и OR (см. раздел 4.3), оператора запятая (см. раздел 4.10) не сохраняются. Кроме того, перегруженные версии операторов `&&` и `||` не поддерживают вычислений по сокращенной схеме. Оба операнда вычисляются всегда.

Поскольку перегруженные версии этих операторов не сохраняют порядок вычисления и (или) не поддерживают вычисления по сокращенной схеме, их перегрузка обычно — плохая идея. Пользователи, вероятно, будут удивлены отсутствием привычных гарантий последовательности вычисления в коде при использовании перегруженной версии одного из этих операторов.

Еще один повод не перегружать операторы запятой и обращения к адресу заключается в том, что, в отличие от большинства операторов, язык сам определяет значение этих операторов, когда они применены к объектам типа класса. Поскольку у этих операторов есть встроенное значение, они обычно не должны перегружаться. Пользователи класса будут удивлены, если они поведут себя не так, как обычно.



Обычно операторы запятая, обращение к адресу, логический оператор AND и OR *не должны* быть перегружены.

Использование определений, совместимых со встроенным смыслом

При разработке класса всегда следует сначала подумать об обеспечиваемых им операциях. Только определившись с необходимыми операциями, следует подумать о том, стоит ли определить некую операцию как обычную функцию или как перегруженный оператор. Те операции, которые логически соответствуют операторам, — это хорошие кандидаты на определение в качестве перегруженных операторов.

- Если класс осуществляет операции ввода и вывода, имеет смысл определить операторы сдвига для совместимости с таковыми у встроенных типов.
- Если класс подразумевает проверку на равенство, определите оператор `operator==`. Если у класса есть оператор `operator==`, то у него обычно должен быть также оператор `operator!=`.
- Если у класса должна быть операция упорядочивания, определите оператор `operator<`. Если у класса есть оператор `operator<`, то у него, вероятно, должны быть все операторы сравнения.
- Тип возвращаемого значения перегруженного оператора обычно должен быть совместимым с таковым у встроенной версии оператора: логические операторы и операторы отношения должны возвращать значение типа `bool`, арифметические операторы должны возвращать значение типа класса, операторы присвоения и составные операторы присвоения должны возвращать ссылку на левый операнд.

Составные операторы присвоения

Операторы присвоения должны вести себя аналогично синтезируемым операторам: после присвоения значения левых и правых operandов должны быть одинаковы, а возвратить оператор должен ссылку на левый operand. Перегруженный оператор присвоения должен обобщить смысл встроенного оператора присвоения, а не переиначивать его.

Внимание! Будьте осторожны при использовании перегруженных операторов

Каждый оператор имеет некий смысл, когда он используется для встроенных типов. Бинарный оператор `+`, например, всегда означает сумму. Вполне логично и удобно применять в классе бинарный оператор `+` для аналогичной функции. Например, библиотечный тип `string`, в соответствии с соглашением, общепринятым для множества

языков программирования, использует оператор + для конкатенации, т.е. добавления содержимого одной строки в другую.

Перегруженные операторы полезней всего тогда, когда смысл встроенного оператора логически соответствует функции текущего класса. Применение перегруженных операторов вместо именованных функций позволяет сделать программы более простыми, естественными и интуитивно понятными. Злоупотребление перегруженными операторами, а также приданье не свойственного им смысла сделает класс неудобным в применении.

На практике вполне очевидные случаи противовесственной перегрузки операторов довольно редки. Например, ни один ответственный программист не переопределил бы оператор operator+ для вычитания. Зато очень часто предпринимаются попытки неким образом приспособить "обычный" оператор, который неприменим к данному классу. Операторы следует использовать только для тех функций, которые будут однозначно поняты пользователями. Оператор с неоднозначным смыслом, например равенство, может быть интерпретирован по-разному.

Если класс обладает арифметическим (см. раздел 4.2) или побитовым (см. раздел 4.8) оператором, то его, как правило, имеет смысл снабдить соответствующими составными операторами. Вполне логично было бы также определить и оператор +=. Само собой разумеется, оператор += должен быть определен так, чтобы он вел себя аналогично встроенным операторам, т.е. осуществлял составное присвоение: сначала сумма (+), а затем присвоение (=).

Выбор обычной функции или члена класса

При проектировании перегруженных операторов необходимо принять решение, должен ли каждый из них быть членом класса или обычной функцией (не членом класса). В некоторых случаях выбора нет; оператор должен быть членом класса. В других случаях можно принять во внимание несколько эмпирических правил, которые помогут принять решение.

Приведенный ниже список критериев может оказаться полезен в ходе принятия решения о том, следует ли сделать оператор функцией-членом класса или обычной функцией.

- Операторы присвоения (=), индексирования ([]), вызова (()) и доступа к члену класса (->) *следует* определять как функции-члены класса.

- Подобно оператору присвоения, составные операторы присвоения *обычно* должны быть членами класса. Но в отличие от оператора присвоения, это не обязательно.

- Другие операторы, которые изменяют состояние своего объекта или жестко связаны с данным классом (например, инкремент, декремент и обращение к значению), обычно должны быть членами класса.

- Симметричные операторы, такие как арифметические, операторы равенства, операторы сравнения и побитовые операторы, лучше определять как обычные функции, а не члены класса.

Разработчики ожидают возможности использовать симметричные операторы в выражениях со смешанными типами. Например, возможности сложить переменные типа `int` и `double`. Сложение симметрично, а потому можно использовать тип как левого, так и правого операнда.

Если необходимо обеспечить подобные выражения смешанного типа, задействующие объекты класса, то оператор должен быть определен как функция, не являющаяся членом класса.

При определении оператора как функции-члена левый операнд должен быть объектом того класса, членом которого является этот оператор. Например:

```
string s = "world";
string t = s + "!";    // ok: const char* можно
добавить к строке
string u = "hi" + s;  // возможна ошибка, если +
будет членом
                           // класса string
```

Если бы оператор `operator+` был членом класса `string`, то первый случай сложения был бы эквивалентен `s.operator+("!"`). Аналогично сложение `"hi" + s` было бы эквивалентно `"hi".operator+(s)`. Однако литерал `"hi"` имеет тип `const char*`, т.е. встроенный тип; у него нет функций-членов.

Поскольку класс `string` определяет оператор `+` как обычную функцию, не являющуюся членом класса, сложение `"hi" + s` эквивалентно вызову `operator+("hi", s)`. Подобно любому вызову функции, каждый из аргументов должен быть преобразован в тип параметра. Единственное требование — по крайней мере один из operandов должен иметь тип класса, а оба операнда могут быть преобразованы в строку.

Упражнения раздела 14.1

Упражнение 14.1. Чем перегруженный оператор отличается от встроенного? В чем перегруженные операторы совпадают со встроенными?

Упражнение 14.2. Напишите объявления для перегруженных операторов ввода, вывода, сложения и составного присвоения для класса `Sales_data`.

Упражнение 14.3. Классы `string` и `vector` определяют перегруженный оператор `==`, применимый для сравнения объектов этих типов. Если векторы `svec1` и `svec2` содержат строки, объясните, какая из версий оператора `==` применяется в каждом из следующих выражений:

- (a) "cobble" == "stone" (b) `svec1[0] == svec2[0]`
- (c) `svec1 == svec2` (d) "svec1[0] == "stone"

Упражнение 14.4. Объясните, должен ли каждый из следующих операторов быть членом класса и почему?

- (a) % (b) %= (c) ++ (d) -> (e) << (f) && (g) == (h)
- ()

Упражнение 14.5. В упражнении 7.40 из раздела 7.5.1 был приведен набросок одного из следующих классов. Какой из перегруженных операторов должен (если должен) предоставить класс.

- (a) `Book` (b) `Date` (c) `Employee`
- (d) `Vehicle` (e) `Object` (f) `Tree`



14.2. Операторы ввода и вывода

Как уже упоминалось, библиотека ИО использует операторы `>>` и `<<` для ввода и вывода соответственно. Сама библиотека ИО определяет версии этих операторов для ввода и вывода данных встроенных типов. Классы, нуждающиеся во вводе и выводе, обычно определяют версии этих операторов для объектов данного класса.



14.2.1. Перегрузка оператора вывода `<<`

Обычно первый параметр оператора вывода является ссылкой на неконстантный объект класса `ostream`. Объект класса `ostream`

неконстантен потому, что запись в поток изменяет его состояние. Параметр является ссылкой потому, что нельзя копировать объект класса `ostream`.

Второй параметр обычно должен быть ссылкой на константу типа класса, объект которого необходимо вывести. Параметр должен быть ссылкой во избежание копирования аргумента. Но он может быть константной ссылкой потому, что вывод объекта обычно не изменяет его.

Для совместимости с другими операторами вывода оператор `operator<<` обычно возвращает свой параметр типа `ostream`.

Оператор вывода класса `Sales_data`

Для примера напишем оператор вывода для класса `Sales_data`:

```
ostream &operator<<( ostream &os, const Sales_data &item) {
    os << item.isbn() << " " << item.units_sold << " "
        << item.revenue << " " << item.avg_price();
    return os;
}
```

За исключением имени эта функция идентична прежней версии функции `print()` (см. раздел 7.1.3). Вывод объекта класса `Sales_data` требует вывода значений всех его трех переменных-членов, а также вычисления средней цены (`average price`). Каждый элемент отделяется пробелом. После вывода значений оператор возвращает ссылку на использованный для этого объект класса `ostream`.

Операторы вывода обеспечивают минимум форматирования

Операторы вывода встроенных типов форматирования практически не обеспечивают. В частности, они не выводят символ новой строки. Пользователи ожидают, что операторы вывода класса будут вести себя так же. Если бы оператор выводил новую строку, то пользователь не смог бы вывести содержимое объекта с описывающим его текстом в одной строке. Оператор вывода, обеспечивающий минимум форматирования, позволяет контролировать подробности вывода пользователям.

Рекомендуем

Обычно операторы вывода должны выводить содержимое объекта с минимальным форматированием. Они не должны выводить новую строку.

Операторы ввода-вывода не должны быть функциями-

членами класса

Операторы ввода и вывода, соответствующие соглашениям библиотеки `iostream`, должны быть обычными функциям, а не членами класса. Эти операторы не могут быть членами нашего класса. Если бы это было так, то левый операнд должен был быть объектом типа нашего класса:

```
Sales_data data;  
data << cout; // если бы оператор operator<<  
// был членом класса Sales_data
```

Если бы эти операторы были членами некоего класса, то они должны были бы быть членами класса `istream` или `ostream`. Но эти классы являются частью стандартной библиотеки, а добавлять члены в библиотечные классы нельзя.

Таким образом, если необходимо определить операторы ввода-вывода для собственного типа, их следует определить как функции, не являющиеся членами класса. Конечно, операторы ввода-вывода обычно должны читать или выводить данные не открытых переменных-членов. Как следствие, операторы ввода-вывода обычно объявляют дружественными (см. раздел 7.2.1).

Упражнения раздела 14.2.1

Упражнение 14.6. Определите оператор вывода для класса `Sales_data`.

Упражнение 14.7. Определите оператор вывода для класса `String`, написанного для упражнений раздела 13.5.

Упражнение 14.8. Определите оператор вывода для класса, который был выбран в упражнении 7.40 раздела 7.5.1.



14.2.2. Перегрузка оператора ввода >>

Обычно первый параметр оператора ввода является ссылкой на поток, из которого осуществляется чтение, а второй параметр — ссылкой на некий неконстантный объект, в который предстоит прочитать данные. Обычно оператор возвращает ссылку на свой поток. Второй параметр не должен быть константным потому, что задачей оператора ввода и является собственно запись данных в этот объект.

Оператор ввода класса Sales_data

В качестве примера напишем оператор ввода для класса Sales_data:

```
istream &operator>>( istream &is, Sales_data &item)
{
    double price; // инициализировать не нужно; читать
    в price
        // прежде, чем использовать
    is >> item.bookNo >> item.units_sold >> price;
    if (is) // проверить успех ввода данных
        item.revenue = item.units_sold * price;
    else
        item = Sales_data(); // ввод неудачен: вернуть
    объект в
        // стандартное состояние
    return is;
}
```

За исключением оператора if это определение подобно прежней функции read() (см. раздел 7.1.3). Оператор if проверяет, было ли чтение успешно. Если произойдет ошибка ввода-вывода, он вернет объект Sales_data в состояние пустого объекта. Это гарантирует корректность состояния объекта.



Операторы ввода должны учитывать возможность неудачи ввода, а операторы вывода об этом могут не заботиться.

Ошибки во время ввода

В операторе ввода возможны следующие ошибки.

- Операция чтения может потерпеть неудачу из-за наличия в потоке данных неподходящего типа. Например, после чтения переменной-члена bookNo оператор ввода подразумевает, что следующие два элемента будут числовыми данными. Если во вводе окажутся не числовые данные, поток будет недопустим и все последующее попытки чтения из него потерпят неудачу.

- Во время любой из операций чтения может встретиться конец файла или произойти другая ошибка потока ввода.

Чтобы не проверять каждую часть прочитанных данных, можно проверить состояние потока в целом и только потом использовать прочитанные данные

```
if (is) // проверить успех ввода данных
    item.revenue = item.units_sold * price;
else
    item = Sales_data(); // ввод неудачен: вернуть
объект в
                                // стандартное состояние
```

При сбое любой из операций чтения значение переменной-члена `price` останется неопределенным. Следовательно, перед ее использованием следует проверить, допустим ли еще поток ввода. Если это так, осуществляется вычисление значения переменной `revenue`. В случае ошибки ничего страшного не произойдет, поскольку будет возвращен пустой объект класса `Sales_data`. Для этого объекту `item` присваивается новый объект класса `Sales_data`, созданный при помощи стандартного конструктора. После этого присвоения переменная-член `bookNo` объекта `item` будет содержать пустую строку, а его переменные члены `revenue` и `units_sold` — нулевое значение.

Возвращение объекта в допустимое состояние особенно важно, если объект мог быть частично изменен прежде, чем произошла ошибка. Например, в данном операторе ввода ошибка могла бы произойти уже после успешного чтения в переменную-член `bookNo`. В результате значения переменных-членов `units_sold` и `revenue` останутся неизменными. Таким образом, новое значение `bookNo` будет связано с данными прежнего объекта.

Оставляя объект в допустимом состоянии, можно в некоторой степени защитить пользователя, который игнорирует возможность ошибки ввода. Объект будет находиться в пригодном для использования состоянии — все его члены окажутся определены. Кроме того, объект не будет вводить в заблуждение — его данные останутся единообразными.

Рекомендуем

Проектируя оператор ввода, очень важно решить, что делать в случае ошибки и как вновь сделать объект доступным.

Оповещение об ошибке

Некоторые операторы ввода нуждаются в дополнительной проверке данных. Например, оператор ввода мог бы проверить соответствие формату данных, читаемых в переменную `bookNo`. В таких случаях оператору ввода возможно понадобится установить флаг состояния потока так, чтобы он означал отказ (см. раздел 8.1.2), хотя с технической точки зрения чтение было успешно. Обычно оператор ввода устанавливает только флаг `failbit`. Флаг `eofbit` подразумевал бы конец файла, а бит `badbit` — нарушение потока. Установку этих флагов лучше оставить библиотеке IO.

Упражнения раздела 14.2.2

Упражнение 14.9. Определите оператор ввода для класса `Sales_data`.

Упражнение 14.10. Опишите поведение оператора ввода класса `Sales_data` при следующем вводе:

(a) 0-201-99999-9 10 24.95 (b) 10 24.95 0-210-99999-9

Упражнение 14.11. Что не так со следующим оператором ввода класса `Sales_data`? Что будет при передаче этому оператору данных предыдущего упражнения?

```
istream& operator>>( istream& in, Sales_data& s) {  
    double price;  
    in >> s.bookNo >> s.units_sold >> price;  
    s.revenue = s.units_sold * price;  
    return in;  
}
```

Упражнение 14.12. Определите оператор ввода для класса, использованного в упражнении 7.40 раздела 7.5.1. Обеспечьте обработку оператором ошибок ввода.

14.3. Арифметические операторы и операторы отношения

Как правило, арифметические операторы и операторы отношения определяют как функции не члены класса, чтобы обеспечить преобразования и для левого, и для правого операнда (см. раздел 7.1.5). Эти операторы не должны изменять состояние любого из операндов, поэтому их параметры обычно являются ссылками на константу.

Обычно арифметический оператор создает новое значение, являющееся

результатом вычисления двух своих операндов. Это значение отлично от каждого из операндов и вычисляется в локальной переменной. Оператор возвращает как результат копию этого локального значения. Классы, определяющие арифметический оператор, определяют также соответствующий составной оператор присвоения. Когда у класса есть два оператора, как правило, эффективней определять арифметический оператор для составного присвоения:

```
// подразумевается, что оба объекта относятся к той же книге
Sales_data
operator+(const Sales_data &lhs, const Sales_data &rhs) {
    Sales_data sum = lhs; // копирование переменных-членов из lhs в sum
    sum += rhs;           // добавить rhs к sum
    return sum;
}
```

Это определение очень похоже на оригинальную функцию `add()` (см. раздел 7.1.3). Значение `lhs` копируется в локальную переменную `sum`. Затем оператор составного присвоения класса `Sales_data` (определенный в разделе 14.4) добавляет значение `rhs` к `sum`. Функция завершает работу, возвращая копию значения переменной `sum`.



Классы, в которых определен арифметический оператор и соответствующий ему составной оператор, обычно реализуют арифметический оператор при помощи составного.

Упражнения раздела 14.3

Упражнение 14.13. Какие еще арифметические операторы (см. табл. 4.1), если таковые вообще есть, должны, по-вашему, поддержать класс `Sales_data`? Определите эти операторы.

Упражнение 14.14. Почему оператор `operator+` эффективней определять как вызывающий оператор `operator+=`, а не наоборот?

Упражнение 14.15. Должен ли класс, выбранный в упражнении 7.40 раздела 7.5.1, определять какие-либо арифметические операторы? Если да,

то реализуйте их. В противном случае объясните, почему нет.



14.3.1. Операторы равенства

Классы языка C++ используют оператор равенства для проверки эквивалентности объектов. Он сравнивает каждую переменную-член обоих объектов и признает их равными, если все значения одинаковы. В соответствии с этой концепцией оператор равенства класса Sales_data должен сравнить переменные bookNo двух объектов, а также значения их остальных переменных.

```
bool operator==(const Sales_data &lhs, const Sales_data &rhs) {
    return lhs.isbn() == rhs.isbn() &&
           lhs.units_sold == rhs.units_sold &&
           lhs.revenue == rhs.revenue;
}
bool operator!=(const Sales_data &lhs, const Sales_data &rhs) {
    return !(lhs == rhs);
}
```

Определение этих функций тривиально. Однако важнее всего принципы, которые здесь используются.

- Если в классе определен оператор, позволяющий выяснить равенство двух объектов данного класса, его функция должна иметь имя operator==. Не стоит изобретать для нее другое имя, поскольку пользователи ожидают, что для сравнения объектов можно использовать именно оператор ==. Кроме того, это гораздо проще, чем каждый раз запоминать новые имена.
- Если в классе определен оператор ==, то два объекта могут содержать одинаковые данные.
 - Обычно оператор равенства должен быть транзитивным, т.е. если оба выражения, a == b и b == c, являются истинными, то a == c тоже должно быть истиной.
 - Если в классе определен оператор operator==, следует также определить и оператор operator!=. Пользователи вполне резонно будут полагать, что если применимо равенство, то применимо и неравенство.

- Определяя операторы равенства и неравенства, почти всегда имеет смысл использовать один из них для создания другого. Один оператор должен фактически сравнивать объекты, а второй — использовать его в своих целях.

Рекомендуем

Классы, в которых определен оператор `operator==`, гораздо проще использовать со стандартной библиотекой. Если оператор `==` определен в классе, то такие алгоритмы к нему можно применять без всякой дополнительной подготовки.

Упражнения раздела 14.3.1

Упражнение 14.16. Определите операторы равенства и неравенства для классов `StrBlob` (см. раздел 12.1.1), `StrBlobPtr` (см. раздел 12.1.6), `StrVec` (см. раздел 13.5) и `String` (см. раздел 13.5).

Упражнение 14.17. Должен ли класс, выбранный в упражнении 7.40 раздела 7.5.1, определять операторы равенства? Если да, то реализуйте их. В противном случае объясните, почему нет.



14.3.2. Операторы отношения

Классы, для которых определен оператор равенства, зачастую (но не всегда) обладают операторами отношения. В частности, это связано с тем, что ассоциативные контейнеры и некоторые из алгоритмов используют оператор меньше (`operator<`).

Обычно операторы отношения должны определять следующее.

1. Порядок отношений, совместимый с требованиями для ключей ассоциативных контейнеров (см. раздел 11.2.2);
2. Отношение, совместимое с равенством, если у класса есть оба оператора. В частности, если два объекта не равны, то один объект должен быть меньше другого.



Вполне резонно предположить, что класс `Sales_data` должен

поддерживать операторы отношения, хотя это и не обязательно. Причины не столь очевидны, поэтому рассмотрим их подробнее.

Можно подумать, что оператор `<` будет определен так же, как функция `compareIsbn()` (см. раздел 11.2.2). Эта функция сравнивала объекты класса `Sales_data` за счет сравнения их ISBN. Хотя функция `compareIsbn()` обеспечивает порядок отношений, что соответствует первому требованию, она возвращает результат, противоречащий определению равенства. В результате она не удовлетворяет второму требованию.

Оператор `==` класса `Sales_data` считает две транзакции с одинаковым ISBN неравными, если у них отличаются значения переменных-членов `revenue` или `units_sold`. Если бы оператор `<` был определен как сравнивающий только значения ISBN, то два объекта с одинаковым ISBN, но разными `units_sold` или `revenue` считались бы неравными, но ни один из объектов не был бы меньше другого. Как правило, если имеются два объекта, ни один из которых не меньше другого, то вполне логично ожидать, что эти объекты равны.

Создается впечатление, что имеет смысл определить оператор `operator<` для сравнения каждой переменной-члена по очереди. Его можно было бы определить так, чтобы при равных `isbn` объекты сравнивались по переменной-члену `units_sold`, а затем `revenue`.

Однако никаких оснований для упорядочивания здесь нет. В зависимости от того, как планируется использовать класс, определить порядок можно сначала на основании переменных `revenue` и `units_sold`. Можно было бы установить, что объекты с меньшим значением переменной `units_sold` были "меньше", чем таковые с большим. Либо можно было бы установить, что объекты с меньшим значением переменной-члена `revenue` "меньше", чем таковые с большим значением.

Для класса `Sales_data` нет единого логического определения значения "меньше". Таким образом, для этого класса лучше вообще не определять оператор `operator<`.

Рекомендуем

Если есть однозначное логическое определение значения "меньше", то классы обычно должны определять оператор `operator<`. Но если у

класса есть также оператор `operator==`, то определяйте оператор `operator<`, только если определения смысла понятий "меньше" и "равно" не противоречат друг другу.

Упражнения раздела 14.3.2

Упражнение 14.18. Определите операторы отношения для классов `StrBlob`, `StrBlobPtr`, `StrVec` и `String`.

Упражнение 14.19. Определяет ли класс, выбранный в упражнении 7.40 раздел 7.5.1, операторы отношения? Если да, то реализуйте их. В противном случае объясните, почему нет.

14.4. Операторы присвоения

Кроме операторов присвоения копии и присваивания при перемещении, которые присваивают один объект типа класса другому объекту того же класса (см. раздел 13.1.2 и раздел 13.6.2), в классе можно определить дополнительные операторы присвоения, позволяющие использовать в качестве правого операнда другие типы.

Например, библиотечный класс `vector`, кроме операторов присвоения копии и присваивания при перемещении, определяет третий оператор присвоения, получающий заключенный в фигурные скобки список элементов (см. раздел 9.2.5). Этот оператор можно использовать следующим образом:

```
vector<string> v;
v = {"a", "an", "the"};
```

Такой оператор можно также добавить в класс `StrVec` (см. раздел 13.5):

```
class StrVec {
public:
    StrVec &operator=
    (std::initializer_list<std::string>);  
    // другие члены, как в разделе 13.5
}
```

Чтобы не отличаться от операторов присвоения для встроенных типов (и уже определенных операторов присвоения копии и присваивания при перемещении), новый оператор присвоения будет возвращать ссылку на левый operand:

```
StrVec &StrVec::operator=(std::initializer_list<string>
```

```

il) {
    // alloc_n_copy() резервирует пространство и
копирует элементы
    // из заданного диапазона
    auto data = alloc_n_copy(il.begin(), il.end());
    free(); // удалить элементы в этом объекте и
освободить пространство
    elements = data.first; // обновить переменные-
члены, чтобы указывать
                           // на новое пространство
    first_free = cap = data.second;
    return *this;
}

```

Подобно операторам присвоения копии и присваивания при перемещении, другие перегруженные операторы присвоения должны освобождать существующие элементы и создавать новые. В отличие от операторов копирования и присваивания при перемещении, этот оператор не должен проверять случай присвоения себя себе. Параметр имеет тип `initializer_list<string>` (см. раздел 6.2.6), а это означает, что объект `il` не может быть тем же объектом, на который указывает указатель `this`.



Операторы присвоения могут быть перегружены. Независимо от типа параметра, операторы присвоения следует определять как функции-члены.

Составные операторы присвоения

Составные операторы присвоения не обязаны быть функциями-членами. Однако все операторы присвоения, включая составные, предпочтительно определять в классе. Для согласованности со встроенными составными операторами присвоения эти операторы должны возвращать ссылку на левый операнд. Например, ниже приведено определение составного оператора присвоения для класса `Sales_data`.

```

// бинарный оператор-член:
// левый operand связан с неявным указателем this
// подразумевается, что оба объекта относятся к той

```

же книге

```
Sales_data& Sales_data::operator+=( const Sales_data
&rhs) {
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}
```



Обычно операторы присвоения и составные операторы присвоения должны быть определены как функции-члены и возвращать ссылку на левый операнд.

Упражнения раздела 14.4

Упражнение 14.20. Определите оператор суммы и составной оператор присвоения для класса `Sales_data`.

Упражнение 14.21. Напишите операторы класса `Sales_data` так, чтобы `+` осуществлял сложение, а оператор `+=` вызывал оператор `+`. Обсудите недостатки этого подхода по сравнению со способом, которым эти операторы были определены в разделах 14.3 и 14.4.

Упражнение 14.22. Определите версию оператора присвоения, способного присвоить строку, представляющую ISBN, объекту класса `Sales_data`.

Упражнение 14.23. Определите в версии класса `StrVec` оператор присвоения для типа `initializer_list`.

Упражнение 14.24. Примите решение, нуждается ли класс из упражнения 7.40 раздела 7.5.1 в операторах копирования и присваивания при перемещении. Если да, то определите эти операторы.

Упражнение 14.25. Реализуйте все остальные операторы присвоения, которые должен определить класс. Объясните, какие типы должны использоваться как операнды и почему.

14.5. Оператор индексирования

Классы, представляющие контейнеры, способные возвращать элементы по позиции, зачастую определяют оператор индексирования `operator[]`.



Оператор индексирования должен быть определен как функция-член класса.

Согласно общепринятому смыслу индексирования, оператор индексирования обычно возвращает ссылку на выбранный элемент. Возвращающий ссылку оператор индексирования применим с обеих сторон оператора присвоения. Следовательно, имеет смысл определить и константную, и неконстантную версии этого оператора. При применении к константному объекту оператор индексирования должен возвращать ссылку на константу, чтобы предотвратить присвоение возвращенному объекту.

Рекомендуем

Если у класса есть оператор индексирования, он обычно должен быть определен в двух версиях: возвращающей простую ссылку и являющуюся константной функцией-членом, а следовательно, возвращающую ссылку на константу.

В качестве примера определим оператор индексирования для класса StrVec (см. раздел 13.5):

```
class StrVec {  
public:  
    std::string& operator[]( std::size_t n)  
    { return elements[ n] ; }  
    const std::string& operator[]( std::size_t n) const  
    { return elements[ n] ; }  
    // другие члены, как в разделе 13.5  
private:  
    std::string *elements; // указатель на первый  
Элемент массива  
};
```

Эти операторы можно использовать таким же образом, как и индексирование вектора или массива. Поскольку оператор индексирования возвращает ссылку на элемент, если объект класса StrVec не константен,

то этому элементу можно присвоить значение; если индексируется константный объект, присвоение невозможно:

```
// svec - объект класса StrVec
const StrVec cvec = svec; // копировать элементы из
svec в cvec
// если у svec есть элементы, выполнить функцию
empty() класса string
// для первого
if (svec.size() && svec[0].empty()) {
    svec[0] = "zero"; // ok: индексирование возвращает
ссылку на строку
    cvec[0] = "Zip"; // ошибка: индексация с vec
возвращает ссылку на
// константу
}
```

Упражнения раздела 14.5

Упражнение 14.26. Определите операторы индексирования для классов `StrVec`, `String`, `StrBlob` и `StrBlobPtr`.

14.6. Операторы инкремента и декремента

Операторы инкремента (`++`) и декремента (`--`) обычно реализуют для классов итераторов. Эти операторы позволяют перемещать итератор с элемента на элемент последовательности. Язык никак не требует, чтобы эти операторы были членами класса. Но поскольку они изменяют состояние объекта, с которым работают, лучше сделать их членами класса.

Для встроенных типов есть префиксные и постфиксные версии операторов инкремента и декремента. Ничего удивительного, что для собственных классов также можно определить префиксные и постфиксные версии этих операторов. Давайте сначала рассмотрим префиксные версии, а затем реализуем постфиксные.

Рекомендуем

Классы, определяющие операторы инкремента или декремента, должны определять как префиксные, так и постфиксные их версии. Обычно эти операторы определяют как функции-члены.

Определение префиксных версий операторов инкремента и декремента

Для иллюстрации операторов инкремента и декремента определим их для класса `StrBlobPtr` (см. раздел 12.1.6):

```
class StrBlobPtr {  
public:  
    // инкремент и декремент  
    StrBlobPtr& operator++(); // префиксные операторы  
    StrBlobPtr& operator--();  
    // другие члены как прежде  
};
```

 Рекомендуем

Чтобы соответствовать встроенным типам, префиксные операторы должны возвращать ссылку на объект после инкремента или декремента.

Операторы инкремента и декремента работают подобным образом — они вызывают функцию `check()` для проверки допустимости объекта класса `StrBlobPtr`. Если это так, то функция `check()` проверяет также допустимость данного индекса. Если функция `check()` не передает исключения, эти операторы возвращают ссылку на свой объект.

В случае инкремента функции `check()` передается текущее значение `curr`. Пока это значение меньше размера основного вектора, функция `check()` завершается нормально. Если значение `curr` находится за концом вектора, функция `check()` передает исключение:

```
// префикс: возвращает ссылку на объект после  
инкремента  
// или декремента  
StrBlobPtr& StrBlobPtr::operator++() {  
    // если curr уже указывает после конца контейнера,  
    // инкремент  
    // невозможен  
    check(curr, "increment past end of StrBlobPtr");  
    ++curr; // переместить текущую позицию вперед  
    return *this;  
}  
StrBlobPtr& StrBlobPtr::operator--() {
```

```

    // если curr равен нулю, то декремент возвратит
    недопустимый индекс
    --curr; // переместить текущую позицию назад
    check( -1, "decrement past begin of StrBlobPtr");
    return *this;
}

```

Оператор декремента уменьшает значение `curr` прежде, чем вызвать функцию `check()`. Таким образом, если значение `curr` (беззнаковое) уже является нулем, передаваемое функции `check()` значение будет наибольшим позитивным значением, представляющим недопустимый индекс (см. раздел 2.1.2).

Дифференциация префиксных и постфиксных операторов

При определении префиксных и постфиксных операторов возникает одна проблема: каждый из них имеет одинаковое имя и получает одинаковое количество параметров того же типа. При обычной перегрузке невозможно отличить префиксную и постфиксную версии оператора.

Для решения этой проблемы постфиксные версии получают дополнительный (неиспользуемый) параметр типа `int`. При использовании постфиксного оператора компилятор присваивает этому параметру аргумент 0. Хотя постфиксная функция вполне может использовать этот дополнительный параметр, как правило, так не поступают. Этот параметр не нужен для работы, обычно выполняемой постфиксным оператором. Его основная задача заключается в том, чтобы отличить определение постфиксной версии функции от префиксной.

Теперь в класс `CheckedPtr` можно добавить постфиксные операторы:

```

class StrBlobPtr {
public:
    // инкремент и декремент
    StrBlobPtr operator++( int );      // постфиксные
    операторы
    StrBlobPtr operator--( int );
    // другие члены как прежде
} ;

```



Для совместимости со встроенными операторами постфиксные операторы

должны возвращать прежнее значение (существовавшее до декремента или инкремента). Оно должно быть возвращено как значение, а не как ссылка.

Постфиксные версии должны запоминать текущее состояние объекта прежде, чем изменять объект:

```
// постфикс:    инкремент/декремент    объекта,    но
возвратить следует
// неизмененное значение
StrBlobPtr StrBlobPtr::operator++(int) {
    // здесь проверка не нужна, ее выполнит префиксный
инкремент
    StrBlobPtr ret = *this; // сохранить текущее
значение
    ++*this; // на один элемент вперед,
проверку // осуществляет оператор
инкремента
    return ret; // возврат сохраненного
значения
}
StrBlobPtr StrBlobPtr::operator--(int) {
    // здесь проверка не нужна, ее выполнит префиксный
декремент
    StrBlobPtr ret = *this; // сохранить текущее
значение
    --*this; // на один элемент назад,
проверку // осуществляет оператор
декремента
    return ret; // возврат сохраненного
значения
}
```

Для выполнения фактического действия каждый из этих операторов вызывает собственную префиксную версию. Например, постфиксный оператор инкремента использует такой вызов префиксного оператора инкремента:

```
+*this
```

Этот оператор проверяет безопасность приращения и либо передает исключение, либо осуществляет приращение значения curr. Если

функция `check()` не передает исключения, постфиксные функции завершают работу, возвращая сохраненные ранее копии значений. Таким образом, после выхода сам объект будет изменен, но возвращено будет первоначальное, не измененное значение.



Поскольку параметр типа `int` не используется, имя ему присваивать не нужно.

Явный вызов постфиксных операторов

Как упоминалось в разделе 14.1, в качестве альтернативы использованию перегруженного оператора в выражении можно вызвать его явно. Если постфиксная версия задействуется при помощи вызова функции, то следует передать значение и для целочисленного аргумента:

```
StrBlobPtr p(a1); // p указывает на вектор в a1
p.operator++(0); // вызов постфиксного оператора
operator++
p.operator++(); // вызов префиксного оператора
operator++
```

Переданное значение обычно игнорируется, но оно позволяет предупредить компилятор о том, что требуется именно постфиксная версия оператора.

Упражнения раздела 14.6

Упражнение 14.27. Добавьте в класс `StrBlobPtr` операторы инкремента и декремента.

Упражнение 14.28. Определите для класса `StrBlobPtr` операторы сложения и вычитания, чтобы они реализовали арифметические действия с указателями (см. раздел 3.5.3).

Упражнение 14.29. Почему не были определены константные версии операторов инкремента и декремента?

14.7. Операторы доступа к членам

Операторы обращения к значению (`*`) и стрелка (`->`) обычно используются в классах, представляющих итераторы, и в классах интеллектуального указателя (см. раздел 12.1). Вполне логично добавить

эти операторы в класс StrBlobPtr:

```
class StrBlobPtr {  
public:  
    std::string& operator*() const {  
        auto p = check(curr, "dereference past end");  
        return (*p)[curr]; // (*p) - вектор, на который  
указывает этот  
                                // объект  
    }  
    std::string* operator->() const {  
        // передать реальную работу оператору обращения к  
значению  
        return &this->operator*();  
    }  
    // другие члены как прежде  
};
```

Оператор обращения к значению проверяет принадлежность curr диапазону, и если это так, то возвращает ссылку на элемент, обозначенный значением curr. Оператор стрелки не делает ничего сам, он вызывает оператор обращения к значению и возвращает адрес возвращенного им элемента.



Оператор стрелка (arrow) должен быть определен как функция-член класса. *Оператор обращения к значению (dereference)* необязательно должен быть членом класса, но, как правило, его тоже определяют как функцию-член.

Следует заметить, что эти операторы определены как константные члены. В отличие от операторов инкремента и декремента, выборка элемента никак не изменяет состояния объекта класса StrBlobPtr. Обратите также внимание на то, что эти операторы возвращают ссылку или указатель на неконстантную строку. Причина этого в том, что объект класса StrBlobPtr, как известно, может быть связан только с неконстантным объектом класса StrBlob (см. раздел 12.1.6).

Эти операторы можно использовать таким же способом, которым используются соответствующие операторы с указателями и итераторами

вектора:

```
StrBlob a1 = {"hi", "bye", "now"};  
StrBlobPtr p(a1); // p указывает на  
вектор в a1  
*p = "okay"; // присвоить первый  
элемент a1  
cout << p->size() << endl; // выводит 4, размер  
первого элемента в a1  
cout << (*p).size() << endl; // эквивалент p-  
>size()
```

Ограничения на возвращаемое значение оператора стрелки

Подобно большинству других операторов (хотя это и плохая идея), оператор `operator*` можно определить как выполняющий некоторые действия по своему усмотрению. Таким образом, оператор `operator*` можно определить как возвращающий, например, фиксированное значение, скажем, `42`, или выводящий содержимое объекта, к которому он применен, или что то еще. Но для перегруженного оператора стрелки это не так. Оператор стрелки никогда не изменяет своего фундаментального назначения: доступа к члену класса. При перегрузке оператора стрелки можно изменить объект, из которого стрелка выбирает определенный член, но нельзя изменить тот факт, что она выбирает член класса.

В коде `point->mem` часть `point` должна быть указателем на объект класса или объектом класса с перегруженным оператором `operator->`. В зависимости от типа части `point` код `point->mem` может быть эквивалентен следующему:

```
(*point).mem; // point - указатель  
встроенного типа  
point.operator() ->mem; // point - объект типа  
класса
```

В противном случае код ошибочен. Таким образом, код `point->mem` выполняется следующим образом.

1. Если `point` — указатель, то применение встроенного оператора стрелки означает эквивалент выражения `(*point).mem`. Указатель обращается к значению члена класса и выбирает его из объекта. Если у типа, на объект которого указывает `point`, нет члена по имени `mem`, то этот код ошибочен.

2. Если `point` — объект класса, в котором определен оператор

`operator->`, то результат вызова `point.operator->()` используется для выбора члена `mem`. Если результат является указателем, то для него выполняется этап 1. Если результат является объектом, класс которого сам обладает перегруженным оператором `operator->()`, то с этим объектом повторяется данный этап. Процесс продолжается до тех пор, пока не будет возвращен указатель на объект с означенным членом или некое другое значение, означающее ошибочность кода.



Перегруженный оператор стрелки *должен* возвращать либо указатель на тип класса, либо объект типа класса, определяющего собственный оператор стрелки.

Упражнения раздела 14.7

Упражнение 14.30. Добавьте операторы обращения к значению и стрелки в класс `StrBlobPtr` и класс `ConstStrBlobPtr` из упражнения 12.22 раздела 12.1.6. Обратите внимание, что операторы класса `ConstStrBlobPtr` должны возвращать константные ссылки, поскольку переменная-член `data` класса `ConstStrBlobPtr` указывает на константный вектор.

Упражнение 14.31. В классе `StrBlobPtr` не определен конструктор копий, оператор присвоения и деструктор. Почему?

Упражнение 14.32. Определите класс, содержащий указатель на класс `StrBlobPtr`. Определите перегруженный оператор стрелки для этого класса.



14.8. Оператор вызова функции

Классы, перегружающие оператор вызова, позволяют использовать объекты этого типа как функции. Поскольку объекты таких классов способны хранить состояние, они могут оказаться существенно гибче обычных функций.

В качестве простого примера рассмотрим структуру `absInt`, обладающую оператором вызова, возвращающим абсолютное значение своего аргумента:

```
struct absInt {  
    int operator()(int val) const {  
        return val < 0 ? -val : val;  
    }  
};
```

Этот класс определяет одну функцию: оператор вызова функции. Этот оператор получает аргумент типа `int` и возвращает абсолютное значение аргумента.

Оператор вызова используется применительно к списку аргументов объекта класса `absInt` способом, который выглядит как вызов функции:

```
int i = -42;  
absInt absObj;           // объект класса с оператором  
вызыва функции  
int ui = absObj(i);     // передача i в  
absObj.operator()
```

Хотя `absObj` — это объект, а не функция, его вполне можно вызвать. При вызове объект выполняет свой перегруженный оператор вызова. В данном случае этот оператор получает значение типа `int` и возвращает его абсолютное значение.



Оператор вызова функции должен быть функцией-членом. Класс может определить несколько версий оператора вызова, каждая из которых должна отличаться количеством или типом параметров.

Объект класса, определяющего оператор вызова, называется *объектом функции* (function object). Такие объекты действуют как функции, поскольку их можно вызвать.

Классы объектов функций с состоянием

У класса объекта функции, как у любого другого класса, могут быть и другие члены, кроме оператора `operator()`. Классы объекта функции зачастую содержат переменные-члены, используемые для настройки действий в операторе вызова.

В качестве примера определим класс, выводящий строковый аргумент. По умолчанию класс будет писать в поток `cout` и выводить пробел после каждой строки. Позволим также пользователям класса предоставлять другой поток для записи и другой разделитель. Этот класс можно определить следующим образом:

```
class PrintString {  
public:  
    PrintString(ostream &o = cout, char c = ' ') :  
        os(o), sep(c) {}  
    void operator()(const string &s) const { os << s  
<< sep; }  
private:  
    ostream &os; // поток для записи  
    char sep; // символ завершения после каждого  
вывода  
};
```

У класса есть конструктор, получающий ссылку на поток вывода, и символ, используемый как разделитель. Как аргументы по умолчанию (см. раздел 6.5.1) для этих параметров используется поток `cout` и пробел. Тело оператора вызова функции использует эти члены при выводе данной строки.

При определении объектов класса `PrintString` можно использовать аргументы по умолчанию или предоставлять собственные значения для разделителя или потока вывода:

```
PrintString printer; // использует аргументы по  
умолчанию; вывод в cout  
printer(s); // выводит s и пробел в cout  
PrintString errors(cerr, '\n');  
errors(s); // выводит s и новую строку в  
cerr
```

Объекты функции обычно используют как аргументы для обобщенных алгоритмов. Например, для вывода содержимого контейнера можно использовать класс `PrintString` и библиотечный алгоритм `for_each()` (см. раздел 10.3.2):

```
for_each(vs.begin(), vs.end(), PrintString(cerr,
'\\n'));
```

Третий аргумент алгоритма `for_each()` является временным объектом типа `PrintString`, инициализируемый потоком `cerr` и символом новой строки. Вызов функции `for_each()` выводит каждый элемент `vs` в поток `cerr`, разделяя их новой строкой.

Упражнения раздела 14.8

Упражнение 14.33. Сколько операндов может иметь перегруженный оператор вызова функции?

Упражнение 14.34. Определите класс объекта функции для выполнения действий условного оператора: оператор вызова этого класса должен получать три параметра. Он должен проверить свой первый параметр и, если эта проверка успешна, возвратить свой второй параметр; в противном случае он должен возвратить свой третий параметр.

Упражнение 14.35. Напишите класс, подобный классу `PrintString`, который читает строку из потока `istream` и возвращает строку, представляющую прочитанное. При неудаче чтения следует возвратить пустую строку.

Упражнение 14.36. Используя класс из предыдущего упражнения, организуйте чтение со стандартного устройства ввода, сохраняя каждую строку в векторе как элемент.

Упражнение 14.37. Напишите класс, проверяющий равенство двух значений. Используйте этот объект и библиотечные алгоритмы для написания кода замены всех экземпляров заданного значения в последовательности.

14.8.1. Лямбда-выражения — объекты функции

В предыдущем разделе объект `PrintString` использовался как аргумент в вызове функции `for_each()`. Это похоже на программу, написанную в разделе 10.3.2, где использовалось лямбда-выражение. Написанное лямбда-выражение компилятор преобразовывает в безымянный объект безымянного класса (см. раздел 10.3.3). Классы, созданные из лямбда-выражения, содержат перегруженный оператор

вызыва функции. Рассмотрим, например, лямбда-выражение, передававшееся как последний аргумент функции `stable_sort()`:

```
// сортировать слова по размеру, поддерживаая
// алфавитный порядок среди
// слов того же размера
stable_sort(words.begin(), words.end(),
            [](const string &a, const string &b)
            { return a.size() < b.size();});
```

Это действует как безымянный объект класса, который выглядел бы примерно так:

```
class ShorterString {
public:
    bool operator()(const string &s1, const string
&s2) const
    { return s1.size() < s2.size(); }
};
```

У этого класса есть один член, являющийся оператором вызова функции, получающим две строки и сравнивающий их длины. Список параметров и тело функции те же, что и у лямбда-выражения. Как уже упоминалось в разделе 10.3.3, по умолчанию лямбда-выражения не могут изменять свои захваченные переменные. В результате по умолчанию оператор вызова функции в классе, созданном из лямбда-выражения, является константной функцией-членом. Если лямбда-выражение объявляется как `mutable`, то оператор вызова не будет константным.

Вызов функции `stable_sort()` можно переписать так, чтобы использовать этот класс вместо лямбда-выражения:

```
stable_sort(words.begin(), words.end(),
ShorterString());
```

Третий аргумент — недавно созданный составной объект класса `ShorterString`. Код в функции `stable_sort()` будет вызывать этот объект каждый раз, когда он сравнивает две строки. При вызове объекта будет выполнено тело его оператора вызова, возвращающего значение `true`, если размер первой строки будет меньше, чем второй.

Классы, представляющие лямбда-выражения с захваченными переменными

Как уже упоминалось, при захвате лямбда-выражением переменной по ссылке разработчик должен сам гарантировать существование переменной, на которую ссылается ссылка, во время выполнения лямбда-выражения

(см. раздел 10.3.3). Поэтому компилятору разрешено использовать ссылку непосредственно, не сохраняя ее как переменную-член в созданном классе.

Переменные, которые захватываются по значению, напротив, копируются в лямбда-выражение (см. раздел 10.3.3). В результате классы, созданные из лямбда-выражений, переменные которых захватываются по значению, имеют переменные-члены, соответствующие каждой такой переменной. У этих классов есть также конструктор для инициализации этих переменных-членов значениями захваченных переменных. В примере раздела 10.3.2 лямбда-выражение использовалось для поиска первой строки, длина которой была больше или равна заданному значению:

```
// получить итератор на первый элемент, размер которого >= sz
auto wc = find_if(words.begin(), words.end(),
                   [sz](const string &a)
```

Созданный класс выглядел бы примерно так:

```
class SizeComp {
    SizeComp(size_t n) : sz(n) {} // параметр для
каждой захваченной
                                    // переменной
    // оператор вызова с тем же типом возвращаемого
значения, параметрами
    // и телом, как у лямбда-выражения
    bool operator()(const string &s) const
    { return s.size() >= sz; }
private:
    size_t sz; // переменная-член для каждой
переменной, захваченной
                                    // по значению
};
```

В отличие от класса `ShorterString`, у этого класса есть переменная-член и конструктор для ее инициализации. У этого синтезируемого класса нет стандартного конструктора; чтобы использовать этот класс, следует передать аргумент:

```
// получить итератор на первый элемент, размер
которого >= sz
auto wc = find_if(words.begin(), words.end(),
SizeComp(sz));
```

У классов, созданных из лямбда-выражения, есть удаленный стандартный конструктор, удаленные операторы присвоения и

стандартный деструктор. Будет ли у класса стандартный или удаленный конструктор копий/перемещения, зависит обычно от способа и типа захватываемых переменных-членов (см. раздел 13.1.6 и раздел 13.6.2).

Упражнения раздела 14.8.1

Упражнение 14.38. Напишите класс, проверяющий соответствие длины заданной строки указанному значению. Используйте такой объект в программе для оповещения о количестве слов во входном файле, имеющих размеры от 1 до 10 включительно.

Упражнение 14.39. Перепишите предыдущую программу так, чтобы сообщать количество слов размером от 1 до 9 и 10 или более.

Упражнение 14.40. Перепишите функцию `biggies()` из раздела 10.3.2 так, чтобы использовать объект функции вместо лямбда-выражения.

Упражнение 14.41. Как по-вашему, существенно ли добавление лямбда-выражений по новому стандарту? Объясните, когда имеет смысл использовать лямбда-выражение, а когда класс вместо него.

14.8.2. Библиотечные объекты функций

Стандартная библиотека определяет набор классов, представляющих арифметические, реляционные и логические операторы. Каждый класс определяет оператор вызова, который применяет одноименный оператор. Например, у класса `plus` есть оператор вызова функции, который применяет оператор `+` к паре operandов; класс `modulus` определяет оператор вызова, применяющий бинарный оператор `%`; класс `equal_to` применяет оператор `==`; и т.д.

Эти классы являются шаблонами, которым передается один тип. Он определяет тип параметра оператора вызова. Например, класс `plus<string>` применяет строковый оператор суммы к объектам класса `string`; у класса `plus<int>` типом operandов будет `int`; класс `plus<Sales_data>` применяет оператор `+` к объектам класса `Sales_data`; и т.д.:

```
plus<int> intAdd;           // объект функции, способный
сложить
                                // два значения типа int
negate<int> intNegate; // объект функции, способный
изменить знак
                                // значения типа int
// использование оператора intAdd::operator( int,
```

```

int) для
    // сложения чисел 10 и 20
    int sum = intAdd(10, 20);           // эквивалент sum
= 30
    sum = intNegate(intAdd(10, 20)); // эквивалент sum
= 30
    // использование оператора intNegate::operator( int )
для создания
    // числа -10 как второго параметра выражения
intAdd::operator( int, int )
    sum = intAdd(10, intNegate(10)); // sum = 0
Эти типы, перечислены в табл. 4.2, определены в заголовке
functional.

```

Таблица 14.2. Библиотечные объекты функций

Арифметические	Реляционные	Логические
plus<Type>	equal_to<Type>	logical_and<Type>
minus<Type>	not_equal_to<Type>	logical_or<Type>
multiplies<Type>	greater<Type>	logical_not<Type>
divides<Type>	greater_equal<Type>	
modulus<Type>	less<Type>	
negate<Type>	less_equal<Type>	

Применение библиотечного объекта функции с алгоритмами

Классы объектов функций, представляющие операторы, зачастую используются для переопределения заданного по умолчанию оператора, используемого алгоритмом. Как уже упоминалось, по умолчанию алгоритмы сортировки используют оператор `operator<` для сортировки последовательности в порядке возрастания. Для сортировки в порядке убывания можно передать объект типа `greater`. Этот класс создает оператор вызова, который вызывает оператор "больше" основного типа элемента. Предположим, например, что `svec` — это вектор типа `vector<string>`:

```

// передает временный объект функции, который
применяет
// оператор > к двум строкам
sort(svec.begin(), svec.end(), greater<string>());

```

Это сортирует вектор в порядке убывания. Третий аргумент — безымянный объект типа `greater<string>`. Когда функция `sort()` сравнивает элементы, вместо оператора `<` типа элемента она применит переданный объект функции `greater`. Этот объект применит оператор `>` к элементам типа `string`.

Одним из важнейших аспектов этих библиотечных объектов функций является то, что библиотека гарантирует их работоспособность с указателями. Помните, что результат сравнения двух несвязанных указателей непредсказуем (см. раздел 3.5.3). Но может понадобиться сортировать вектор указателей на основании их адреса в памяти. Хотя сделать это самостоятельно непросто, вполне можно применить один из библиотечных объектов функции:

```
vector<string *> nameTable; // вектор указателей
// ошибка: указатели в nameTable не связаны,
результат < непредсказуем
sort( nameTable.begin(), nameTable.end(),
      []( string *a, string *b) { return a < b; } );
// ok: библиотека гарантирует, что less для типов
указателя определен
sort( nameTable.begin(), nameTable.end(),
      less<string*>() );
```

Стоит также обратить внимание на то, что ассоциативные контейнеры используют для упорядочивания своих элементов объект типа `less<key_type>`. В результате можно определить набор (`set`) указателей или использовать указатель как ключ в карте (`map`) без необходимости определять тип `less` самостоятельно.

Упражнения раздела 14.8.2

Упражнение 14.42. Используя библиотечные объекты и адаптеры функций, определите объекты для:

- (a) Подсчета количеств значений больше 1024
- (b) Поиска первой строки, не равной `rooh`
- (c) Умножения всех значений на 2

Упражнение 14.43. Используя библиотечные объекты функций, определите, делимо ли переданное значение типа `int` на некий элемент в контейнере целых чисел.

14.8.3. Вызываемые объекты и тип `function`

В языке C++ есть несколько видов вызываемых объектов: функции и указатели на функции, лямбда-выражения (см. раздел 10.3.2), объекты, созданные функцией `bind()` (см. раздел 10.3.4), и классы с перегруженным оператором вызова функции.

Подобно любому другому объекту, у вызываемого объекта есть тип. Например, у каждого лямбда-выражения есть собственный уникальный (безымянный) тип класса. Типы функций и указателей на функции зависят от типа возвращаемого значения, типа аргумента и т.д.

Однако два вызываемых объекта с разными типами могут иметь ту же *сигнатуру вызова* (call signature). Сигнатурой вызова определяет тип возвращаемого значения вызываемого объекта и тип (типы) аргумента, которые следует передать при вызове. Сигнатура вызова соответствует типу функции. Например:

```
int( int, int)
```

Функция этого типа получает два числа типа `int` и возвращает значение типа `int`.

Разные типы могут иметь одинаковую сигнатуру вызова

Иногда необходимо использовать несколько вызываемых объектов с одинаковой сигнатурой вызова, как будто это тот же тип. Рассмотрим, например, следующие разные типы вызываемых объектов:

```
// обычная функция
int add( int i, int j) { return i + j; }
// лямбда-выражение, создающее безымянный класс
объекта функции
auto mod = [ ](int i, int j) { return i % j; };
// класс объекта функции
struct div {
    int operator()(int denominator, int divisor) {
        return denominator / divisor;
    }
};
```

Каждый из этих вызываемых объектов применяет арифметическую операцию к своим параметрам. Даже при том, что у каждого из них разный тип, сигнатура вызова у них одинакова:

```
int( int, int)
```

Эти вызываемые объекты можно использовать для написания простого калькулятора. Для этого следует определить *таблицу функций* (function table), хранящую "указатели" на вызываемые объекты. Когда программе понадобится выполнить некую операцию, она просмотрит таблицу и найдет соответствующую функцию.

В языке C++ таблицы функций довольно просто реализовать при помощи карт (map). В данном случае как ключ используем строку, соответствующую символу оператора; значение будет функцией, реализующей этот оператор. При необходимости выполнить заданный оператор индексируется карта и осуществляется вызов возвращенного элемента. Если бы все эти функции были автономными и необходимо было использовать только парные операторы для типа int, то карту можно было бы определить так:

```
// сопоставляет оператор с указателем на функцию,
получающую два целых
// числа и возвращающую целое число
map<string, int(*)(int, int)> binops;
```

Указатель add можно поместить в карту binops следующим образом:

```
// ok: add - указатель на функцию соответствующего
типа
binops.insert({ "+" , add}); // {"+", add} - пара
```

раздел 11.2.3

Но сохранить в карте binops объекты mod или div не получится:

```
binops.insert({ "%" , mod}); // ошибка: mod - не
указатель на функцию
```

Проблема в том, что mod — это лямбда-выражение, и у каждого лямбда-выражения есть собственный тип класса. Этот тип не соответствует типу значений, хранимых в карте binops.

Библиотечный тип function



Эту проблему можно решить при помощи нового библиотечного типа function, определенного в заголовке functional; возможные операции с типом function приведены в табл. 14.3.

Таблица 14.3. Операции с типом function

f — пустой объект класса function, способный хранить
--

<code>function<T> f;</code>	вызываемые объекты с сигнатурой вызова, эквивалентной типу функции <code>T</code> (т.е. <code>T</code> — это <code>retType(args)</code>)
<code>function<T> f(nullptr);</code>	Явное создание пустого объекта класса <code>function</code>
<code>function<T> f(obj);</code>	Сохранение копии вызываемого объекта <code>obj</code> в объекте <code>f</code>
<code>f</code>	Когда <code>f</code> используется как условие; оно истинно, если содержит вызываемый объект, и ложно в противном случае
<code>f(args)</code>	Вызывает объект <code>f</code> с передачей аргументов <code>args</code>
Типы, определенные как члены шаблона <code>function<T></code>	
<code>result_type</code>	Тип возвращаемого значения объекта функции этого типа
<code>argument_type</code> <code>first_argument_type</code> <code>second_argument_type</code>	Типы, определяемые, когда у типа <code>T</code> есть один или два аргумента. Если у типа <code>T</code> есть один аргумент, то <code>argument_type</code> — синоним его типа. Если у типа <code>T</code> два аргумента, то <code>first_argument_type</code> и <code>second_argument_type</code> — синонимы их типов

Тип `function` — это шаблон. Подобно другим шаблонам, при создании его экземпляра следует указать дополнительную информацию. В данном случае этой информацией является сигнатура вызова объекта, который сможет представлять данный конкретный тип `function`. Как и у других шаблонов, этот тип определяют в угловых скобках:

```
function<int( int, int )>
```

Здесь был объявлен тип `function`, способный представлять вызываемые объекты, возвращающие целочисленный результат и имеющие два параметра типа `int`. Этот тип можно использовать для представления любого из типов приложения калькулятора:

```
function<int( int, int )> f1 = add; // указатель на функцию
```

```
function<int( int, int )> f2 = div(); // объект класса объекта функции
```

```
function<int( int, int )> f3 = [ ]( int i, int j ) // лямбда-выражение
```

```
                { return i * j; };
```

```
cout << f1( 4, 2 ) << endl; // выводит 6
```

```
cout << f2( 4, 2 ) << endl; // выводит 2
```

```
cout << f3( 4, 2 ) << endl; // выводит 8
```

Теперь карту можно переопределить, используя тип `function`:

```
// таблица вызываемых объектов,
```

```
// соответствующих всем бинарным операторам
```

```

    // все вызываемые объекты должны получать по два
    int и возвращать int
    // элемент может быть указателем на функцию,
    объектом функции или
    // лямбда-выражением
    map<string, function<int( int, int )>> binops;

```

В эту карту можно добавить каждый из вызываемых объектов приложения, будь то указатель на функцию, лямбда-выражение или объект функции:

```

map<string, function<int( int, int )>> binops = {
    { "+", add},                                // указатель на функцию
    { "-", std::minus<int>() }, // объект библиотечной
    // функции
    { "/", div( )},                                // пользовательский
    // объект функции
    { "*", []( int i, int j) { return i * j; } }, // безымянное
    // лямбда-выражение
    { "%", mod} };                                // именованный объект
    // лямбда-выражения

```

В карте пять элементов. Хотя все лежащие в основе вызываемые объекты имеют различные типы, каждый из них можно хранить в общем типе `function<int(int, int)>`.

Как обычно, при индексировании карты возвращается ссылка на ассоциированное значение. При индексировании карты `binops` возвращается ссылка на объект типа `function`. Тип `function` перегружает оператор вызова. Этот оператор вызова получает собственные аргументы и передает их хранимому вызываемому объекту:

```

binops[ "+" ]( 10, 5 ); // вызов add( 10, 5 )
binops[ "-" ]( 10, 5 ); // использует оператор; вызов
объекта minus<int>
binops[ "/" ]( 10, 5 ); // использует оператор; вызов
объекта div
binops[ "*" ]( 10, 5 ); // вызов объекта лямбда-функции
binops[ "%" ]( 10, 5 ); // вызов объекта лямбда-функции

```

Здесь происходит вызов каждой из операций, хранимых в карте `binops`. В первом вызове возвращаемый элемент является указателем на

функцию, указывающим на функцию add(). Вызов binops["+"](10, 5) использует этот указатель для вызова функции add с передачей ей значений 10 и 5. Следующий вызов, binops["-"], возвращает объект класса function, хранящий объект типа std::minus<int>. Затем можно вызвать оператор этого объекта и других.

Перегруженные функции и тип function

Нельзя непосредственно хранить имя перегруженной функции в объекте типа function:

```
int add( int i, int j ) { return i + j; }
Sales_data add( const Sales_data&, const Sales_data& );
map<string, function<int( int, int )>> binops;
binops.insert( { "+", add } ); // ошибка: какой именно add?
```

Один из способов разрешения двусмыслинности подразумевает хранение указателя на функцию (см. раздел 6.7) вместо имени функции:

```
int (*fp)( int, int ) = add; // указатель на версию add,
                                // получающую два int
binops.insert( { "+", fp } ); // ok: fp указывает на правую версию add
```

В качестве альтернативы для устранения неоднозначности можно использовать лямбда-выражение:

```
// ok: использование лямбда-выражения
// для устранения неоднозначности при
// выборе используемой версии add
binops.insert( { "+", [ ]( int a, int b ) { return add( a, b ); } } );
```

Вызов в теле лямбда-выражения передает два целых числа. Этому вызову может соответствовать только та версия функции add(), которая получает два целых числа, а следовательно, эта функция и применяется при выполнении лямбда-выражения.



Класс function в новой библиотеке никак не связан с классами

`unary_function` и `binary_function`, которые были частью прежних версий библиотеки. Эти классы были заменены более общей функцией `bind()` (см. раздел 10.3.4).

Упражнения раздела 14.8.3

Упражнение 14.44. Напишите собственную версию простого калькулятора, способного выполнять бинарные операции.



14.9. Перегрузка, преобразование и операторы

В разделе 7.5.4 упоминалось, что неявный конструктор, который может быть вызван с одним аргументом, определяет неявное преобразование. Такие конструкторы преобразовывают объект типа аргумента в тип класса. Можно также определить преобразование из типа класса. Для этого нужно определить оператор преобразования. Конструкторы преобразования и операторы преобразования определяют *преобразования типа класса* (class-type conversion). Такие преобразования называются также *пользовательскими преобразованиями* (user-defined conversion).

14.9.1. Операторы преобразования

Оператор преобразования (conversion operator) — это специальный вид функции-члена класса. Общий синтаксис функции преобразования имеет следующий вид:

```
operator тип( ) const;
```

где *тип* — это имя типа. Операторы преобразования могут быть определены для любого типа (кроме `void`), который может быть типом возвращаемого значения функции (см. раздел 6.1). Преобразование в тип массива или функции недопустимо. Однако преобразование в тип указателя на данные или функцию, а также ссылочные типы вполне возможны.

У операторов преобразования нет явно заданного типа возвращаемого значения и нет параметров, их следует определять как функции-члены. Операции преобразования обычно не должны изменять преобразуемый объект. В результате операторы преобразования обычно определяют как константные члены.



Функция преобразования должна быть функцией-членом, у нее не определен тип возвращаемого значения и пустой список параметров. Функция обычно должна быть константой.

Определение класса с оператором преобразования

Для примера определим небольшой класс, представляющий целое число в диапазоне от 0 до 255:

```
class SmallInt {  
public:  
    SmallInt(int i = 0): val(i) {  
        if (i < 0 || i > 255)  
            throw std::out_of_range("Bad SmallInt value");  
    }  
    operator int() const { return val; }  
private:  
    std::size_t val;  
};
```

Класс `SmallInt` определяет преобразования в и из своего типа. Конструктор преобразует значения арифметического типа в тип `SmallInt`. Оператор преобразования преобразует объекты класса `SmallInt` в тип `int`:

```
SmallInt si;
si = 4; // неявно преобразует 4 в SmallInt, а затем
        // вызывает SmallInt::operator=
si + 3; // неявно преобразует si в int с
последующим целочисленным
        // суммированием
```

Хотя компилятор применяет только одно пользовательское преобразование за раз (см. раздел 4.11.2), неявное пользовательское преобразование можно предварить или сопроводить стандартным (встроенным) преобразованием (см. раздел 4.11.1). В результате конструктору `SmallInt` можно передать любой арифметический тип. Точно так же можно использовать оператор преобразования для преобразования объекта класса `SmallInt` в `int`, а затем преобразовать полученное значение типа `int` в другой арифметический тип:

```
// аргумент типа double преобразуется в int с
использованием
// встроенного преобразования
SmallInt si = 3.14; // вызов конструктора
SmallInt(int)
// оператор преобразования класса SmallInt
преобразует si в int
si + 3.14; // int преобразуется в double с
использованием встроенного
        // преобразования
```

Поскольку операторы преобразования применяются неявно, нет никакого способа передать аргументы этим функциям. Следовательно, операторы преобразования не могут быть определены как получающие параметры. Хотя функция преобразования не определяет тип возвращаемого значения, каждая из них должна возвратить значение соответствующего типа:

```
class SmallInt;
operator int(SmallInt&); // ошибка: не член класса
class SmallInt {
public:
```

```
    int operator int() const; // ошибка: тип
возвращаемого значения
operator int( int = 0) const; // ошибка:
список параметров
operator int*() const { return 42; } // ошибка: 42
не указатель
};
```

Внимание! Не злоупотребляйте функциями преобразования

Как и в случае с перегруженными операторами, разумное использование функций преобразования помогает существенно упростить работу разработчика класса и сделать полученный класс удобным в применении. Однако здесь есть две потенциальные ловушки: определение слишком большого количества функций преобразования может привести к неоднозначности кода, а некоторые преобразования могут оказаться скорее вредными, чем полезными.

Для примера рассмотрим класс Date, представляющий данные о дате. Вполне очевидно, что имеет смысл предоставить способ преобразования объекта класса Date в объект типа int. Но какое значение должна возвращать функция преобразования? Она могла бы возвратить десятичное представление года, месяца и дня. Например, 30 июля 1989 года могло бы быть представлено как значение 19800730 типа int. В качестве альтернативы оператор преобразования мог бы возвращать целое число, соответствующее количеству дней, начиная с некоторой эпохальной даты. Счетчик мог бы считать дни с 1 января 1970 года или некой другой отправной точки. У обоих преобразований есть желаемое свойство, что более поздние даты соответствуют большим целым числам, что может быть очень полезно.

Проблема в том, что нет единого и полного соответствия между объектом типа Date и значением типа int. В таких случаях лучше не определять оператор преобразования. Вместо него класс должен определить один или несколько обычных членов, чтобы извлекать эту информацию в различных форматах.

Операторы преобразования могут привести к удивительным результатам

На практике классы редко предоставляют операторы преобразования. Пользователи, вероятней всего, будут просто удивлены, случись преобразование автоматически, без помощи явного преобразования. Но из

этого эмпирического правила есть одно важное исключение: преобразование в тип `bool` является вполне общепринятым для классов.

По прежним версиям стандарта перед классами с преобразованием в тип `bool` стояла проблема: поскольку тип `bool` арифметический, объект этого типа, допускающего преобразование в тип `bool`, применим в любом контексте, где ожидается арифметический тип. Такие преобразования могут происходить весьма удивительными способами. В частности, если бы у класса `istream` было преобразование в тип `bool`, то следующий код вполне компилировался бы:

```
int i = 42;
cin << i; // этот код был бы допустим, если бы
преобразование
// в тип bool не было явным!
```

Эта программа пытается использовать оператор вывода для входного потока. Для класса `istream` оператор `<<` не определен, поэтому такой код безусловно ошибочен. Но этот код мог бы использовать оператор преобразования в тип `bool`, чтобы преобразовать объект `cin` в `bool`. Полученное значение типа `bool` было бы затем преобразовано в тип `int`, который вполне применим как левый operand встроенной версии оператора сдвига влево. В результате преобразованное значение типа `bool` (1 или 0) было бы сдвинуто влево на 42 позиции.



Явный оператор преобразования

Чтобы предотвратить подобные проблемы, новый стандарт вводит **явный оператор преобразования** (*explicit conversion operator*):

```
class SmallInt { public:
    // компилятор не будет автоматически применять это
    преобразование
    explicit operator int() const { return val; }
    // другие члены как прежде
};
```

Подобно явным конструкторам (см. раздел 7.5.4), компилятор не будет (обычно) использовать явный оператор преобразования для неявных преобразований:

```
SmallInt si = 3; // ok: конструктор класса SmallInt
не является явным
```

```
si + 3; // ошибка: нужно неявное преобразование, но
оператор int
        // является явным
static_cast<int>(si) + 3; // ok: явный запрос
преобразования
```

Если оператор преобразования является явным, такое преобразование вполне можно осуществить. Но за одним исключением такое приведение следует осуществлять явно.

Исключение состоит в том, что компилятор применит явное преобразование в выражении, используемом как условие. Таким образом, явное преобразование будет использовано неявно для преобразования выражения, используемого как:

- условие оператора `if`, `while` или `do`;
- выражение условия в заголовке оператора `for`;
- operand логического оператора `NOT (!)`, `OR (||)` или `AND (&&)`;
- выражение условия в условном операторе `(?:)`.

Преобразование в тип `bool`

В прежних версиях библиотеки типы ввода-вывода определяли преобразование в тип `void*`. Это было сделано во избежание проблем, описанных выше. По новому стандарту библиотека ввода-вывода определяет вместо этого явное преобразование в тип `bool`.

Всякий раз, когда потоковый объект используется в условии, применяется оператор `operator bool()`, определенный для типов ввода-вывода. Например:

```
while (std::cin >> value)
```

Условие в операторе `while` выполняет оператор ввода, который читает в переменную `value` и возвращает объект `cin`. Для обработки условия объект `cin` неявно преобразуется функцией преобразования `istream::operator bool()`. Эта функция возвращает значение `true`, если флагом состояния потока `cin` является `good` (см. раздел 8.1.2), и `false` в противном случае.



Рекомендуем

Преобразование в тип `bool` обычно используется в условиях. В результате оператор `operator bool` обычно должен определяться как явный.

Упражнения раздела 14.9.1

Упражнение 14.45. Напишите операторы преобразования для преобразования объекта класса `Sales_data` в значения типа `string` и `double`. Какие значения, по-вашему, должны возвращать эти операторы?

Упражнение 14.46. Объясните, является ли определение этих операторов преобразования класса `Sales_data` хорошей идеей и должны ли они быть явными.

Упражнение 14.47. Объясните различие между этими двумя операторами преобразования:

```
struct Integral {  
    operator const int();  
    operator int() const;  
};
```

Упражнение 14.48. Должен ли класс из упражнения 7.40 раздела 7.5.1 использовать преобразование в тип `bool`. Если да, то объясните почему и укажите, должен ли оператор быть явным. В противном случае объясните, почему нет.

Упражнение 14.49. Независимо от того, хороша ли эта идея, определите преобразование в тип `bool` для класса из предыдущего упражнения.



14.9.2. Избегайте неоднозначных преобразований

Если у класса есть один или несколько операторов преобразования, важно гарантировать наличие только одного способа преобразования из типа класса в необходимый тип. Если будет больше одного способа осуществления преобразования, то будет весьма затруднительно написать однозначный код.

Есть два случая, когда возникает несколько путей осуществления преобразования. Первый — когда два класса обеспечивают взаимное преобразование. Например, взаимное преобразование осуществляется тогда, когда класс A определяет конструктор преобразования, получающий объект класса B, а класс B определяет оператор преобразования в тип A.

Второй случай возникновения нескольких путей преобразования — определение нескольких преобразований в и из типов, которые сами связаны преобразованиями. Самый очевидный пример — встроенные арифметические типы. Каждый класс обычно должен определять не больше одного преобразования в или из арифметического типа.



ВНИМАНИЕ

Обычно не следует определять классы со взаимными преобразованиями или определять преобразования в или из арифметических типов.

Распознавание аргумента и взаимные преобразования

В следующем примере определены два способа получения объекта класса A из B: либо при помощи оператора преобразования класса B, либо при помощи конструктора класса A, получающего объект класса B:

```
// обычно взаимное преобразование между двумя
тиปами - плохая идея
struct B;
struct A {
    A() = default;
    A(const B&); // преобразует B в A
    // другие члены
};
```

```

struct B {
    operator A() const; // тоже преобразует B в A
    // другие члены
};

A f( const A& );
A a = f(b); // ошибка неоднозначности:
f(B::operator A())
    // или f(A::A(const B&))

```

Поскольку существуют два способа получения объекта класса A из B, компилятор не знает, какой из них использовать; поэтому вызов функции f() неоднозначен. Для получения объекта класса B этот вызов может использовать конструктор класса A или оператор преобразования класса B, преобразующий объект класса B в A. Поскольку обе эти функции одинаково хороши, вызов неоднозначен и ошибочен.

Если этот вызов необходим, оператор преобразования или конструктор следует вызвать явно:

```

A a1 = f(b.operator A()); // ok: использовать
оператор преобразования B
A a2 = f(A(b));           // ok: использовать
конструктор класса A

```

Обратите внимание: нельзя решить неоднозначность при помощи приведения — у самого приведения будет та же двусмысленность.

Двусмысленность и множественность путей преобразования во встроенные типы

Двусмысленность возникает также в случае, когда класс определяет несколько преобразований в (или из) типы, которые сами связываются преобразованиями. Самый простой и наглядный пример (а также особенно проблематичный) — это когда класс определяет конструкторы преобразования в или из более, чем один арифметический тип.

Например, у следующего класса есть конструкторы преобразования из двух разных арифметических типов и операторы преобразования в два разных арифметических типа:

```

struct A {
    A(int = 0);                                // обычно плохая идея
иметь два
    A(double);                                 // преобразования из
арифметических типов
    operator int() const;                      // обычно плохая идея

```

```

иметь два
operator double() const; // преобразования в
арифметические типы
// другие члены
};

void f2( long double );
A a;
f2( a ); // ошибка неоднозначности: f( A::operator
int() )
// или f( A::operator double() )
long lg;
A a2( lg ); // ошибка неоднозначности: A::A( int ) или
A::A( double )

```

В вызове функции `f2()` ни одно из преобразований не соответствует точно типу `long double`. Но для его получения применимо любое преобразование, сопровождаемое стандартным преобразованием. Следовательно, никакое из преобразований не лучше другого, значит, вызов неоднозначен.

Возникает та же проблема, что и при попытке инициализации объекта `a2` значением типа `long`. Ни один из конструкторов не соответствует точно типу `long`. Каждый требовал преобразования аргумента прежде, чем использовать конструктор.

- Стандартное преобразование `long` в `double`, затем `A(double)`.
- Стандартное преобразование `long` в `int`, затем `A(int)`.

Эти последовательности преобразований равнозначны, поэтому вызов неоднозначен.

Вызов функции `f2()` и инициализация объекта `a2` неоднозначны, поскольку у необходимых стандартных преобразований одинаковый ранг (см. раздел 6.6.1). Когда используется пользовательское преобразование, ранг стандартного преобразования, если таковые вообще имеются, позволяет выбрать наилучшее соответствие:

```

short s = 42;
// преобразование short в int лучше, чем short в
double
A a3( s ); // используется A::A( int )

```

В данном случае преобразование `short` в `int` предпочтительней, чем `short` в `double`. Следовательно, объект `a3` создается с использованием конструктора `A::A(int)`, который запускается для преобразования

значения `s`.



Когда используются два пользовательских преобразования, ранг стандартного преобразования, если таковое вообще имеется, используется для выбора наилучшего соответствия.

Перегруженные функции и конструкторы преобразования

Выбор из нескольких возможных преобразований еще более усложняется, когда происходит вызов перегруженной функции. Если два или более преобразования обеспечивают подходящее соответствие, то преобразования считаются одинаково хорошими.

Например, могут возникнуть проблемы неоднозначности, когда перегруженные функции получают параметры, отличающиеся типами классов, которые определяют те же конструкторы преобразования:

Внимание! Преобразования и операторы

Корректная разработка перегруженных операторов, конструкторов преобразования и функций преобразования для класса требует большой осторожности. В частности, если в классе определены и операторы преобразования, и перегруженные операторы, вполне возможны неоднозначные ситуации. Здесь могут пригодиться следующие эмпирические правила.

- Никогда не создавайте взаимных преобразований типов. Другими словами, если класс `Foo` имеет конструктор, получающий объект класса `Bar`, не создавайте в классе `Bar` оператор преобразования для типа `Foo`.
- Избегайте преобразований во встроенные арифметические типы. Но если преобразование в арифметический тип необходимо, то придется учесть следующее.
 - Не создавайте перегруженных версий тех операторов, которые получают аргументы арифметических типов. Если пользователи используют эти операторы, функция преобразования преобразует объект данного типа, а затем применит встроенный оператор.
 - Не создавайте функций преобразования больше, чем в один арифметический тип. Позвольте осуществлять преобразования в другие арифметические типы стандартным функциям преобразования.

Самое простое правило: за исключением явного преобразования в тип

`bool`, избегайте создания функций преобразования и ограничьте неявные конструкторы теми, которые безусловно необходимы.

```
struct C {  
    C( int );  
    // другие члены  
};  
struct D {  
    D( int );  
    // другие члены  
};  
void manip( const C& );  
void manip( const D& );  
manip( 10 ); // ошибка неоднозначности: manip( C( 10 ) )  
или manip( D( 10 ) )
```

Здесь у структур `C` и `D` есть конструкторы, получающие значение типа `int`. Для версий функции `manip()` подходит любой конструктор. Следовательно, вызов неоднозначен: он может означать преобразование `int` в `C` и вызов первой версии `manip()` или может означать преобразование `int` в `D` и вызов второй версии.

Вызывающая сторона может устраниТЬ неоднозначность при явном создании правильного типа:

```
manip( C( 10 ) ); // ok: вызов manip( const C& )
```



Необходимость в использовании конструктора или приведения для преобразования аргумента при обращении к перегруженной функции — это признак плохого проекта.

Перегруженные функции и пользовательские преобразования

Если при вызове перегруженной функции два (или больше) пользовательских преобразования обеспечивают подходящее соответствие, они считаются одинаково хорошими. Ранг любых стандартных преобразований, которые могли бы (или не могли) быть обязательными, не рассматривается. Необходимость встроенного преобразования также рассматривается, только если набор перегруженных версий может быть подобран и использован той же функцией преобразования.

Например, вызов функции `manip()` был бы неоднозначен, даже если бы один из классов определил конструктор, который требовал бы для аргумента стандартного преобразования:

```
struct E {  
    E(double);  
    // другие члены  
};  
void manip2(const C&);  
void manip2(const E&);  
// ошибка неоднозначности: применимы два разных  
пользовательских  
// преобразования  
manip2(10);           //          manip2(C(10))      или  
manip2(E(double(10)))
```

В данном случае у класса `C` есть преобразование из типа `int` и у класса `E` есть преобразование из типа `double`. Для вызова `manip2(10)` подходят обе версии функции `manip2()`:

- Версия `manip2(const C&)` подходит потому, что у класса `C` есть конструктор преобразования, получающий тип `int`. Этот конструктор точно соответствует аргументу.
- Версия `manip2(const E&)` подходит потому, что у класса `E` есть конструктор преобразования, получающий тип `double` и возможность использовать стандартное преобразование для преобразования аргумента типа `int`, чтобы использовать этот конструктор преобразования.

Поскольку вызовы перегруженных функций требуют *разных* пользовательских преобразований друг от друга, этот вызов неоднозначен. В частности, даже при том, что один из вызовов требует стандартного преобразования, а другой является точным соответствием, компилятор все равно отметит этот вызов как ошибку.



Ранг дополнительного стандартного преобразования (если оно есть) при вызове перегруженной функции имеет значение, только если подходящие функции требуют того же пользовательского преобразования. Если необходимы разные пользовательские преобразования, то вызов неоднозначен.

Упражнения раздела 14.9.2

Упражнение 14.50. Представьте возможные последовательности преобразований типов для инициализации объектов `ex1` и `ex2`. Объясните, допустима ли их инициализация или нет.

```
struct LongDouble {  
    LongDouble( double = 0.0 );  
    operator double();  
    operator float();  
};  
LongDouble IdObj;  
int ex1 = IdObj;  
float ex2 = IdObj;
```

Упражнение 14.51. Представьте последовательности преобразования (если они есть), необходимые для вызова каждой версии функции `calc()`, и объясните, как подбирается наилучшая подходящая функция.

```
void calc( int );  
void calc( LongDouble );  
double dval;  
calc( dval ); // которая calc()?
```



14.9.3. Подбор функций и перегруженные операторы

Перегруженные операторы — это перегруженные функции. При выявлении, который из встроенных или перегруженных операторов применяется для данного выражения, используется обычный подбор функции (см. раздел 6.4). Однако, когда в выражении используется функция оператора, набор функций-кандидатов шире, чем при вызове функций, использующих оператор вызова. Если объект `a` имеет тип класса, то выражение `a sym b` может быть следующим:

- `operator sym(b);` // класс `a` содержит оператор `sym` как функцию-член
- `operator sym(a, b);` // оператор `sym` — обычная функция

В отличие от обычных вызовов функции, нельзя использовать форму вызова для различия функции-члена или не члена класса.

Когда используется перегруженный оператор с операндом типа класса,

функции-кандидаты включают обычные версии, не являющиеся членами класса этого оператора, а также его встроенные версии. Кроме того, если левый операнд имеет тип класса, определенные в нем перегруженные версии оператора (если они есть) также включаются в набор кандидатов.

Когда вызывается именованная функция, функции-члены и не члены класса с тем же именем *не* перегружают друг друга. Перегрузки нет потому, что синтаксис, используемый для вызова именованной функции, различает функции-члены и не члены класса. При вызове через объект класса (или ссылку, или указатель на такой объект) рассматриваются только функции-члены этого класса. При использовании в выражении перегруженного оператора нет никакого способа указать на использование функции-члена или не члена класса. Поэтому придется рассматривать версии и функции-члены, и не члены класса.



Набор функций-кандидатов для используемого в выражении оператора может содержать функции-члены и не члены класса.

Определим, например, оператор суммы для класса `SmallInt`:

```
class SmallInt {  
    friend  
        SmallInt operator*( const SmallInt&, const  
SmallInt&);  
    public:  
        SmallInt( int = 0 ); //  
преобразование из int  
        operator int() const { return val; } //  
преобразование в int  
    private:  
        std::size_t val;  
};
```

Этот класс можно использовать для суммирования двух объектов класса `SmallInt`, но при попытке выполнения смешанных арифметических операций возникнет проблема неоднозначности:

```
SmallInt s1, s2;  
SmallInt s3 = s1 + s2; // использование  
перегруженного оператора +
```

```
int i = s3 + 0;           // ошибка: неоднозначность
```

Первый случай суммирования использует перегруженную версию оператора + для суммирования двух значений типа SmallInt. Второй случай неоднозначен, поскольку 0 можно преобразовать в тип SmallInt и использовать версию оператора + класса SmallInt либо преобразовать объект s3 в тип int и использовать встроенный оператор суммы для типа int.



Предоставление функции преобразования в арифметический тип и перегруженных операторов для того же типа может привести к неоднозначности между перегруженными и встроенными операторами.

Упражнения раздела 14.9.3

Упражнение 14.52. Какой из операторов operator+, если таковые вообще имеются, будет выбран для каждого из следующих выражений суммы? Перечислите функции-кандидаты, подходящие функции и преобразования типов для аргументов каждой подходящей функции:

```
struct LongDouble {  
    // оператор-член operator+ только для демонстрации;  
    // обычно он не является членом класса  
    LongDouble operator+(const SmallInt&); // другие члены как в р. 14.9.2  
};  
LongDouble operator+(LongDouble&, double);  
SmallInt si;  
LongDouble ld;  
ld = si + ld;  
ld = ld + si;
```

Упражнение 14.53. С учетом определения класса SmallInt определите, допустимо ли следующее выражение суммы. Если да, то какой оператор суммы используется? В противном случае, как можно изменить код, чтобы сделать его допустимым?

```
SmallInt s1;  
double d = s1 + 3.14;
```

Резюме

Перегруженный оператор должен либо быть членом класса, либо иметь по крайней мере один операнд типа класса. У перегруженных операторов должно быть то же количество operandов, порядок и приоритет, как у соответствующего оператора встроенного типа. Когда оператор определяется как член класса, его неявный указатель `this` связан с первым operandом. Операторы присвоения, индексирования, вызова функции и стрелки должны быть членами класса.

Объекты классов, которые перегружают оператор вызова функции, `operator()` называются "объектами функций". Такие объекты зачастую используются в комбинации со стандартными алгоритмами. Лямбда-выражения — это отличный способ определения простых классов объектов функций.

Класс может определить преобразования в или из своего типа, которые будут использованы автоматически. Неявные конструкторы, которые могут быть вызваны с одним аргументом, определяют преобразования из типа параметра в тип класса; операторы неявного преобразования определяют преобразования из типа класса в другие типы.

Термины

Объект функции (function object). Объект класса, в котором определен перегруженный оператор вызова. Объекты функций применяются там, где обычно ожидаются функции.

Оператор преобразования (conversion operator). Оператор преобразования — это функция-член, которая осуществляет преобразование из типа класса в другой тип. Операторы преобразования должны быть константными членами их класса. Такие функции не получают параметров и не имеют типа возвращаемого значения. Они возвращают значение типа оператора преобразования. То есть оператор `operator int` возвращает тип `int`, оператор `operator string` — тип `string` и т.д.

Перегруженный оператор (overloaded operator). Функция, переопределяющая значение одного из встроенных операторов. Функция перегруженного оператора имеет имя `operator` с последующим определяемым символом. У перегруженных операторов должен быть по крайней мере один operand типа класса. У перегруженных операторов тот же приоритет, порядок и количество operandов, что и у их встроенных

аналогов.

Пользовательское преобразование (user-defined conversion). Синоним термина преобразование типа класса.

Преобразование типа класса (class-type conversion). Преобразования в или из типа класса определяются конструкторами и операторами преобразования соответственно. Неявные конструкторы, получающие один аргумент, определяют преобразование из типа аргумента в тип класса. Операторы преобразования определяют преобразования из типа класса в заданный тип.

Сигнатура вызова (call signature). Представляет интерфейс вызываемого объекта. Сигнатура вызова включает тип возвращаемого значения и заключенный в круглые скобки разделяемый запятыми список типов аргументов.

Таблица функций (function table). Контейнер, как правило, карта или вектор, содержащий значения, позволяющие выбрать и выполнить функцию во время выполнения.

Шаблон функции (function template). Библиотечный шаблон, способный представить любой вызываемый тип.

Явный оператор преобразования (explicit conversion operator). Оператор преобразования с предшествующим ключевым словом `explicit`. Такие операторы используются для неявных преобразований только в определенных условиях.

Глава 15

Объектно-ориентированное программирование

Объектно-ориентированное программирование основано на трех фундаментальных концепциях: абстракция данных, наследование и динамическое связывание.

Наследование и динамическое связывание рационализируют программы двумя способами: они упрощают создание новых классов, которые подобны, но не идентичны другим классам, а также облегчают написание программы, позволяя игнорировать незначительные различия в подобных классах.

При создании большинства приложений используются одинаковые принципы, которые различаются лишь способами их реализации. Например, рассматриваемый для примера книжный магазин мог бы применять различные системы тарификации для разных книг. Некоторые книги можно было бы продавать лишь по фиксированной цене, а для других применить гибкую систему скидок. Можно было бы предоставлять скидку тем покупателям, которые покупают несколько экземпляров книги. Скидку можно было бы также предоставить на несколько первых экземпляров, а для остальных оставить полную цену. *Объектно-ориентированное программирование* (Object-Oriented Programming, или ООП) — это наилучший способ создания приложений такого типа.



15.1. Краткий обзор ООП

Ключевыми концепциями объектно-ориентированного программирования являются абстракция данных, наследование и динамическое связывание. Используя абстракцию данных, можно определить классы, отделяющие интерфейс от реализации (см. главу 7). Наследование позволяет определять классы, моделирующие отношения между подобными типами. Динамическое связывание позволяет использовать объекты этих типов, игнорируя незначительные различия между ними.

Наследование

Связанные *наследованием* (inheritance) классы формируют иерархию. В корне иерархии обычно находится *базовый класс* (base class), от которого прямо или косвенно происходят другие классы. Эти унаследованные классы известны как *производные классы* (derived class). В базовом классе определяют те члены, которые будут общими у всех типов в иерархии. В производных классах определяются те члены, которые будут специфическими для данного производного класса.

Для моделирования разных стратегий расценок определим класс *Quote*, который будет базовым классом нашей иерархии. Объект класса *Quote* представит книгу без скидок. От него унаследуем второй класс, *Bulk_quote*, представляющий книги, которые могут быть проданы со скидкой за опт.

У этих классов будут две функции-члена.

- Функция *isbn()* будет возвращать ISBN. Она никак не зависит от специфических особенностей производных классов; поэтому будет определена только в классе *Quote*.

- Функция *net_price(size_t)* будет возвращать цену при покупке определенного количества экземпляров книги. Эта операция специфична для типа; классы *Quote* и *Bulk_quote* определят собственные версии этой функции.

В языке C++ базовый класс отличает функции, специфические для типа, от тех, которые предполагается наследовать в производных классах без изменений. Те функции, которые производные классы должны определять самостоятельно, базовый класс определяет как *virtual*. Исходя из этого, класс *Quote* можно первоначально написать так:

```
class Quote {  
public:  
    std::string isbn() const;  
    virtual double net_price( std::size_t n ) const;  
};
```

Производный класс должен указать класс (классы), который он намеревается унаследовать. Для этого используется находящийся после двоеточия *список наследования класса* (class derivation list), представляющий собой разделяемый запятыми список базовых классов, у каждого из которых может быть необязательный спецификатор доступа:

```
class Bulk_quote : public Quote { // Bulk_quote  
наследуется от Quote  
public:
```

```
    double net_price( std::size_t ) const override;
};
```

Поскольку класс `Bulk_quote` использует в списке наследования спецификатор `public`, его объекты можно использовать так, как будто они являются объектами класса `Quote`.

Тело производного класса должно включать объявления всех *виртуальных функций* (*virtual function*), которые он намеревается определить для себя. Производный класс может включить в эти функции ключевое слово `virtual`, но не обязательно. По причинам, рассматриваемым в разделе 15.3, новый стандарт позволяет производному классу явно указать, что функция-член предназначена для *переопределения* (`override`) унаследованной виртуальной функции. Для этого после списка ее параметров располагают ключевое слово `override`.

Динамическое связывание

Динамическое связывание (*dynamic binding*) позволяет взаимозаменяемо использовать тот же код для обработки объектов как типа `Quote`, так и `Bulk_quote`. Например, следующая функция выводит общую стоимость при покупке заданного количества экземпляров указанной книги:

```
// вычислить и отобразить цену за указанное
// количество экземпляров
// с применением всех скидок
double print_total( ostream &os,
                     const Quote &item, size_t n) {
    // в зависимости от типа, связанного с параметром
    item объекта,
    // вызвать функцию Quote::net_price() или
    Bulk_quote::net_price()
    double ret = item.net_price( n );
    os << "ISBN: " << item.isbn() // вызов
    Quote::isbn()
    << "# sold: " << n << " total due: " << ret <<
    endl;
    return ret;
}
```

Эта функция довольно проста — она выводит результаты вызова функций `isbn()` и `net_price()` для своего параметра и возвращает значение, вычисленное вызовом функции `net_price()`.

Однако у этой функции есть два интересных момента: по описанным в разделе 15.2.3 причинам, поскольку параметр `item` является ссылкой на тип `Quote`, эту функцию можно вызвать как для объекта класса `Quote`, так и для объекта класса `Bulk_quote`. По причинам, описанным в разделе 15.2.1, поскольку функция `net_price()` является виртуальной, а функция `print_total()` вызывает ее через ссылку, выполняемая версия функции `net_price()` будет зависеть от типа объекта, переданного функции `print_total()`:

```
// basic имеет тип Quote; bulk имеет тип Bulk_quote  
print_total(cout, basic, 20); // вызов версии  
net_price() класса Quote  
print_total(cout, bulk, 20); // вызов версии  
net_price()  
                                // класса Bulk_quote
```

Первый вызов передает функции `print_total()` объект класса `Quote`. Когда функция `print_total()` вызовет функцию `net_price()`, будет выполнена ее версия из класса `Quote`. В следующем вызове, где аргумент имеет тип `Bulk_quote`, будет выполнена версия функции `net_price()` из класса `Bulk_quote` (применяющая скидку). Поскольку решение о выполняемой версии зависит от типа аргумента, оно может быть принято до времени выполнения. Поэтому динамическое связывание иногда называют *привязкой во время выполнения* (run-time binding).



В языке C++ динамическое связывание происходит тогда, когда обращение к виртуальной функции осуществляется при помощи ссылки (или указателя) на базовый класс.

15.2. Определение базовых и производных классов

Во многих, но не всех случаях базовые и производные классы определяются, как и другие классы, но отличия все же имеются. В этом разделе рассматриваются основные возможности, используемые при определении классов, связанных наследованием.



15.2.1. Определение базового класса

Для начала завершим определение класса `Quote`:

```
class Quote {
public:
    Quote() = default; // = default см. раздел 7.1.4
    Quote( const std::string &book, double
sales_price):
        bookNo(book), price(sales_price) { }
    std::string isbn() const { return bookNo; }
    // возвращает общую цену за определенное
количество проданных
    // экземпляров, а различные системы скидок
определяют и
    // применяют производные классы
    virtual double net_price( std::size_t n) const
    { return n * price; }
    virtual ~Quote() = default; // динамическое
связывание для
                                // деструктора
private:
    std::string bookNo; // идентификатор экземпляра
protected:
    double price = 0.0; // стандартная цена (без
скидки)
};
```

Новым в этом классе являются использование ключевого слова `virtual` в функции `net_price()` и деструкторе, а также спецификатора доступа `protected`. Виртуальные деструкторы

рассматриваются в разделе 15.7.1, а пока следует заметить, что корневой класс иерархии наследования почти всегда определяет виртуальный деструктор.



Базовые классы обычно должны определять виртуальный деструктор. Виртуальные деструкторы необходимы, даже если они не делают ничего.

Функции-члены и наследование

Производные классы наследуют члены своих базовых классов. Но производный класс должен быть в состоянии обеспечить собственное определение таких зависимых от типа операций, как `net_price()`. В таких случаях производный класс должен *переопределить* унаследованное от базового класса определение, обеспечив собственное определение.

В языке C++ базовый класс должен отличать функции, которые предполагается переопределить в производных классах, от тех, которые производные классы, вероятно, наследуют без изменений. Функции, переопределение которых предполагается в производных классах, базовый класс определяет как `virtual`. Когда вызов виртуальной функции происходит *через указатель или ссылку*, он будет привязан динамически. В зависимости от типа объекта, с которым связана ссылка или указатель, будет выполнена версия базового или одного из его производных классов.

Базовый класс определяет, что функция-член должна быть привязана динамически, предваряя ее объявление ключевым словом `virtual`. Любая нестатическая функция-член (см. раздел 7.6), кроме конструктора, может быть виртуальной. Ключевое слово `virtual` присутствует только в объявлении в классе и не может использоваться в определении функции вне тела класса. Функция, объявленная виртуальной в базовом классе, неявно является виртуальной и в производных классах. Более подробная информация о виртуальных функциях приведена в разделе 15.3.

Функции-члены, которые не объявлены как `virtual`, распознаются во время компиляции, а не во время выполнения. Это именно то поведение, которое необходимо для функции `isbn()`. Она не зависит от подробностей производного типа и ведет себя одинаково как с объектами класса `Quote`, так и `Bulk_quote`. В нашей иерархии наследования будет только одна версия функции `isbn()`. Таким образом, не будет никаких

вопросов относительно выполняемой версии функции `isbn()` при вызове.

Управление доступом и наследование

Производный класс наследует члены, определенные в его базовом классе. Но функции-члены производного класса не обязаны обращаться к членам, унаследованным от базового класса. Подобно любому другому коду, использующему базовый класс, производный класс может обращаться к открытым членам своего базового класса, но не может обратиться к закрытым членам. Но иногда у базового класса могут быть члены, которые следует позволить использовать в производных классах, но все же запретить доступ к ним другим пользователям. В определении таких членов используется спецификатор доступа `protected`.

Класс `Quote` ожидает, что его производные классы определят собственную функцию `net_price()`. Для этого им потребуется доступ к члену `price`. В результате класс `Quote` определяет эту переменную-член как `protected`. Производные классы получат доступ к переменной `bookNo` таким же образом, как и обычные пользователи, — при вызове функции `isbn()`. Следовательно, переменная-член `bookNo` останется закрытой и недоступной классам, производным от класса `Quote`. Более подробная информация о защищенных членах приведена в разделе 15.5.

Упражнения раздела 15.2.1

Упражнение 15.1. Что такое виртуальный член класса?

Упражнение 15.2. Чем спецификатор доступа `protected` отличается от `private`?

Упражнение 15.3. Определите собственные версии класса `Quote` и функции `print_total()`.



15.2.2. Определение производного класса

Производный класс должен определить, от какого класса (классов) он происходит. Для этого используется находящийся после двоеточия *список наследования класса* (class derivation list), представляющий собой разделяемый запятыми список имен определенных ранее классов. Каждому имени базового класса может предшествовать необязательный спецификатор доступа: `public`, `protected` или `private`.

Производный класс должен объявить каждую унаследованную функцию-член, которую он намеревается переопределить. Поэтому класс `Bulk_quote` должен включать функцию-член `net_price()`:

```
class Bulk_quote : public Quote { // Bulk_quote
    // происходит от Quote
    Bulk_quote() = default;
    Bulk_quote( const std::string&, double,
    std::size_t, double);
    // переопределить базовую версию и реализовать
    политику
    // скидок при оптовых закупках
    double net_price(std::size_t) const override;
private:
    std::size_t min_qty = 0; // минимальная покупка
    для скидки
    double discount = 0.0; // доля применяемой
    скидки
};
```

Класс `Bulk_quote` унаследовал функцию `isbn()`, а также переменные-члены `bookNo` и `price` из своего базового класса `Quote`. Он определяет собственную версию функции `net_price()` и имеет две дополнительные переменные-члена — `min_qty` и `discount`, которые определяют минимальное количество экземпляров и скидку, применяемую при его покупке.

Более подробная информация об используемых в списке наследования спецификаторах доступа приведена в разделе 15.5, а пока достаточно знать, что спецификатор доступа определяет, разрешено ли пользователям

производного класса знать, что он унаследован от базового класса.

При открытом наследовании открытые члены базового класса становятся частью интерфейса производного. Кроме того, объект открытого производного типа можно привязать к указателю или ссылке на базовый тип. Поскольку в списке наследования использован спецификатор `public`, интерфейс класса `Bulk_quote` неявно содержит функцию `isbn()`, объект класса `Bulk_quote` можно использовать там, где ожидается указатель или ссылка на объект класса `Quote`.

Большинство классов непосредственно происходит только от одного базового класса. Эта форма наследования, известная как "одиночное наследование", и является темой данной главы. В разделе 18.3 будут описаны классы, у которых в списке наследования больше одного базового класса.

Виртуальные функции в производном классе

Производные классы часто, но не всегда, переопределяют унаследованные виртуальные функции. Если производный класс не переопределяет виртуальную функцию своего базового класса, то, подобно любому другому члену, производный класс наследует версию, определенную в его базовом классе.



Производный класс может применять к переопределяемым функциям ключевое слово `virtual`, но не обязательно. По причинам, рассматриваемым в разделе 15.3, новый стандарт позволяет производному классу явно указывать, что функция-член предназначена для переопределения унаследованной виртуальной функции. Для этого применяется спецификатор `override` в определении после списка параметров, либо после ключевого слова `const`, либо квалификатора ссылки, если член класса константен (см. раздел 7.1.2), или ссылки на функцию (см. раздел 13.6.3).

Объекты производного класса и преобразование производного в базовый

Объект производного класса состоит из несколькими частей: нестатических членов, определенных в самом производном классе, а также объекта, состоящего из нестатических членов каждого его базового класса, от которых он происходит. Таким образом, объект класса `Bulk_quote`

будет содержать четыре части данных: переменные-члены bookNo и price, унаследованные от класса Quote, и переменные-члены min_qty и discount, определенные в классе Bulk_quote.

Хотя стандарт не определяет расположение в памяти производных объектов, объект Bulk_quote можно считать состоящим из двух частей (рис. 15.1).

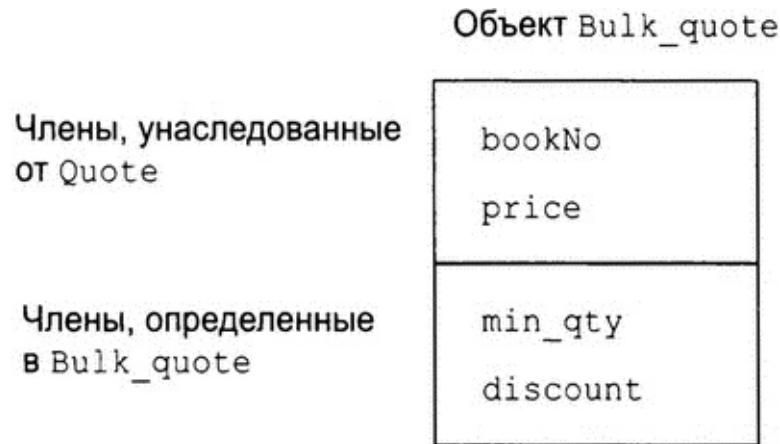


Рис. 15.1. Концептуальная структура объекта класса Bulk_quote



Базовые и производные части объекта вовсе не обязательно будут располагаться рядом. Рис. 15.1 — это концептуальное, не физическое представление работы класса.

Поскольку производная часть объекта соответствует его базовому классу (классам), объект производного типа можно использовать так, как будто это объект его базового класса (классов). В частности, ссылку или указатель на базовый класс можно связать с частью базового класса производного объекта.

```
Quote item;           // объект базового типа
Bulk_quote bulk;     // объект производного типа
Quote *p = &item;      // p указывает на объект Quote
p = &bulk;            // p указывает на часть bulk
объекта Quote
Quote &r = bulk;      // r связан с частью bulk объекта
Quote
```

Это преобразование обычно называют *преобразованием производного в*

базовый (derived-to-base conversion). Подобно любому другому преобразованию, компилятор применяет его неявно (см. раздел 4.11).

Факт неявного преобразования производного в базовый означает возможность использования объекта производного типа или ссылки на него там, где нужна ссылка на базовый тип. Точно так же можно использовать указатель на производный тип там, где требуется указатель на базовый тип.



Факт наличия в объекте производного класса частей объектов его базовых классов является основой работы наследования.

Конструкторы производного класса

Хотя объект производного класса содержит члены, унаследованные им от базового, он не может инициализировать их непосредственно. Как и любой другой код, создающий объект базового класса, производный класс должен использовать конструктор базового класса для инициализации своей части базового класса.



Каждый класс сам контролирует инициализацию своих членов.

Часть базового класса объекта, наряду с переменными-членами производного класса, инициализируется на этапе инициализации конструктора (см. раздел 7.5.1). Аналогично инициализации переменных-членов, для передачи аргументов конструктору базового класса конструктор производного класса использует свой список инициализации. Рассмотрим конструктор `Bulk_quote()` с четырьмя параметрами:

```
Bulk_quote( const std::string& book, double p,
             std::size_t qty, double disc) :
    Quote(book, p), min_qty(qty), discount(disc) { }
    // как прежде
};
```

Для инициализации переменных-членов конструктору класса `Quote` передаются его первые два параметра (представляющие ISBN и цену). Этот

конструктор инициализирует базовую часть класса `Bulk_quote` (т.е. переменные-члены `bookNo` и `price`). Когда (пустое) тело конструктора класса `Quote` закончит работу, часть базового класса создаваемого объекта будет инициализирована. Затем инициализируются прямые переменные-члены `min_qty` и `discount`. И наконец, выполняется (пустое) тело конструктора класса `Bulk_quote`.

Подобно переменной-члену, если не определено иное, базовая часть производного объекта инициализируется по умолчанию. Чтобы использовать другой конструктор базового класса, следует предоставить список инициализации конструктора, используя имя базового класса, сопровождаемое, как обычно, заключенным в скобки списком аргументов. Эти аргументы используются для выбора конкретного конструктора базового класса для инициализации базовой части объекта производного класса.



Сначала инициализируются члены базового класса, а затем члены производного класса в порядке их объявления.

Использование членов базового класса из производного

Производный класс может обращаться к открытым и защищенным членам своего базового класса:

```
// если приобретено достаточно количество
экземпляров,
// использовать цену со скидкой
double Bulk_quote::net_price(size_t cnt) const {
    if (cnt >= min_qty)
        return cnt * (1 - discount) * price;
    else
        return cnt * price;
}
```

Эта функция вычисляет цену со скидкой: если приобретенное количество экземпляров превышает значение переменной `min_qty`, к цене (`price`) применяется скидка (`discount`).

Более подробная информация об областях видимости приведена в разделе 15.6, а пока достаточно знать, что область видимости производного

класса вкладывается в область видимости его базового класса. В результате нет никакого различия между тем, как член производного класса использует члены, определенные в его собственном классе (например, `min_qty` и `discount`), и как он использует члены, определенные в его базовом классе (например, `price`).

Ключевая концепция. Соблюдение интерфейса базового класса

Важно понимать, что каждый класс определяет собственный интерфейс. Для взаимодействия с объектом типа класса следует использовать интерфейс этого класса, даже если он — часть базового класса в объекте производного.

В результате конструкторы производного класса не могут непосредственно инициализировать члены своего базового класса. Тело конструктора производного класса может присваивать значения его открытых или защищенных членов базового класса. Хотя он *может* присвоить значения этим членам, обычно это *не применяется*. Как и любой другой пользователь базового класса, производный класс должен соблюдать интерфейс своего базового класса, используя для инициализации своих унаследованных членов его конструктор.

Наследование и статические члены

Если в базовом классе определен статический (`static`) член (см. раздел 7.6), для всей иерархии существует только один его экземпляр. Независимо от количества классов, производных от базового класса, существовать будет только один экземпляр каждого статического члена.

```
class Base {  
public:  
    static void statmem( );  
};  
class Derived : public Base {  
    void f( const Derived& );  
};
```

Статические члены подчиняются обычным правилам управления доступом: если член класса объявлен в базовом классе закрытым, производные классы не получат к нему доступа. Когда статический член класса доступен, к нему можно обращаться как из базового, так и из производного класса:

```
void Derived::f( const Derived &derived_obj ) {
```

```

Base::statmem( );           // ok: statmem( ) определена в
Base
Derived::statmem( );       // ok: Derived наследует
statmem( )
// ok: объект производного класса применим для
доступа к
// статическому члену базового
derived_obj.statmem( );   // доступ в объекте класса
Derived
statmem( );                // доступ в объекте этого
класса
}

```

Объявления производных классов

Производный класс объявляется как любой другой класс (см. раздел 7.3.3). Объявление содержит имя класса, но не включает его список наследования:

```

class Bulk_quote : public Quote; // ошибка: здесь
не может быть списка
                                                // наследования
class Bulk_quote;                         // ok: правильный
способ объявления
                                                // производного
класса

```

Задача объявления в том, чтобы сообщить о существовании имени и какую сущность он обозначает: класс, функцию или переменную. Список наследования и все другие подробности определения должны присутствовать в теле класса.

Классы, используемые как базовые

Класс должен быть определен, а не только объявлен, прежде чем его можно будет использовать как базовый класс:

```

class Quote; // объявлен, но не определен
// ошибка: класс Quote следует определить
class Bulk_quote : public Quote { ... };

```

Причина этого ограничения очевидна: каждый производный класс содержит и может использовать члены, унаследованные от его базового класса. Чтобы использовать эти члены, производный класс должен знать, что они из себя представляют. Одним из следствий этого правила является

невозможность наследования класса от себя самого.

Базовый класс сам может быть производным классом:

```
class Base { /* ... */ };
class D1: public Base { /* ... */ };
class D2: public D1 { /*... */ };
```

В этой иерархии класс `Base` является *прямым базовым* (direct base class) для класса `D1` и *косвенным базовым* (indirect base class) для класса `D2`. Прямой базовый класс указывают в списке наследования. Косвенный базовый класс наследуется производным через его прямой базовый класс.

Каждый класс наследует все члены своего прямого базового класса. Большинство производных классов наследует члены своего прямого базового класса. Члены прямого базового класса включают унаследованные из его базового класса и т.д. по цепи наследования. Фактически самый последний производный объект содержит часть его прямого базового класса и каждого из его косвенных базовых классов.

Предотвращение наследования



Иногда определяют класс, от которого не следует получать другие производные классы. Либо может быть определен класс, который не предусматривается как подходящий на роль базового. По новому стандарту можно воспрепятствовать использованию класса как базового, расположив за его именем спецификатор `final`:

```
class NoDerived final { /* */ }; // класс NoDerived
                                // не может быть
базовым
class Base { /* */ };
// класс Last финальный; нельзя наследовать класс
Last
class Last final : Base { /* */ }; // класс Last не
может быть базовым
class Bad : NoDerived { /* */ }; // ошибка: класс
NoDerived финальный
class Bad2 : Last { /* */ };      // ошибка: класс
Last финальный
```

Упражнения раздела 15.2.2

Упражнение 15.4. Какие из следующих объявлений (если они есть) некорректны? Объясните, почему.

```
class Base { ... };
(a) class Derived : public Derived { ... };
(b) class Derived : private Base { ... };
(c) class Derived : public Base;
```

Упражнение 15.5. Напишите собственную версию класса Bulk_quote.

Упражнение 15.6. Проверьте свою функцию print_total() из упражнения раздела 15.2.1, передав ей объекты класса Quote и Bulk_quote.

Упражнение 15.7. Определите класс, реализующий ограниченную стратегию скидок, которая применяет скидку только к покупкам до заданного предела. Если количество экземпляров превышает этот предел, к остальным применяется обычная цена.



15.2.3. Преобразования и наследование



Понимание того, как происходит преобразование типов между базовыми и производными классами, очень важно для освоения принципов объектно-ориентированного программирования на языке C++.

Обычно ссылку или указатель можно связать только с тем объектом, тип которого либо совпадает с типом ссылки или указателя (см. раздел 2.3.1 и раздел 2.3.2), либо допускает константное преобразование в него (см. раздел 4.11.2). Классы, связанные наследованием, являются важным исключением: с объектом производного типа можно связать указатель или ссылку на тип базового класса. Например, ссылку `Quote&` можно использовать для обращения к объекту `Bulk_quote`, а адрес объекта `Bulk_quote` можно сохранить в указателе `Quote*`.

У факта возможности привязки ссылки (или указателя) на тип базового класса к объекту производного есть очень важное следствие: при использовании ссылки (или указателя) на тип базового класса неизвестен

фактический тип объекта, с которым он связан. Этот объект может быть как объектом базового класса, так и производного.



Подобно встроенным указателям, классы интеллектуальных указателей (см. раздел 12.1) обеспечивают преобразование производного в базовый, позволяя хранить указатель на объект производного типа в интеллектуальном указателе на базовый.



Статический и динамический типы

При использовании связанных наследованием типов нередко приходится отличать *статический тип* (static type) переменной или выражения от *динамического типа* (dynamic type) объекта, который представляет выражение. Статический тип выражения всегда известен на момент компиляции — это тип, с которым переменная объявляется или возвращает выражение. Динамический тип — это тип объекта в области памяти, которую представляет переменная или выражение. Динамический тип не может быть известен во время выполнения.

Рассмотрим пример, когда функция `print_total()` вызывает функцию `net_price()` (см. раздел 15.1):

```
double ret = item.net_price( n );
```

Известно, что статическим типом параметра `item` является `Quote&`. Динамический тип зависит от типа аргумента, с которым связан параметр `item`. Этот тип не может быть известен, пока не произойдет вызов во время выполнения. Если функции `print_total()` передать объект класса `Bulk_quote`, то статический тип параметра `item` будет отличаться от его динамического типа. Как уже упоминалось, статический тип параметра `item` — это `Quote&`, но в данном случае динамическим типом будет `Bulk_quote`.

Динамический тип выражения, которое не является ни ссылкой, ни указателем, всегда будет совпадать со статическим типом этого выражения. Например, переменная типа `Quote` всегда будет объектом класса `Quote`; нельзя сделать ничего, что изменит тип объекта, которому соответствует эта переменная.



Крайне важно понять, что статический тип указателя или ссылки на базовый класс может отличаться от его динамического типа.

Не существует неявного преобразования из базового типа в производный...

Преобразование из производного в базовый существует благодаря тому, что каждый объект производного класса содержит часть базового класса, с которой и могут быть связаны указатели или ссылки на тип базового класса. Для объектов базового класса подобной гарантии нет. Объект базового класса может существовать либо как независимый объект, либо как часть объекта производного класса. У объекта базового класса, не являющегося частью объекта производного, есть только те члены, которые определены базовым классом; в нем не определены члены производного класса.

Поскольку объект базового класса может быть, а может и не быть частью производного объекта, нет никаких автоматических преобразований из базового класса в класс (классы), производный от него:

```
Quote base;
Bulk_quote* bulkP = &base;           // ошибка: нельзя
преобразовать базовый в
                                         // производный
Bulk_quote& bulkRef = base;          // ошибка: нельзя
преобразовать базовый в
                                         // производный
```

Если бы эти присвоения были допустимы, то можно было бы попытаться использовать указатель bulkP или ссылку bulkRef для доступа к членам, которые не существуют в объекте base.

Немного удивительно то, что невозможно преобразование из базового в производный, даже когда с объектом производного класса связан указатель или ссылка на базовый класс:

```
Bulk_quote bulk;
Quote * itemP = &bulk;           // ok: динамический тип
Bulk_quote
Bulk_quote * bulkP = itemP;      // ошибка: нельзя
```

преобразовать базовый в

// производный

У компилятора нет никакого способа узнать (во время компиляции), что некое преобразование окажется безопасно во время выполнения. Компилятор рассматривает только статические типы указателей или ссылок, определяя допустимость преобразования. Если у базового класса есть одна или несколько виртуальных функций, для запроса преобразования, проверяемого во время выполнения, можно использовать оператор `dynamic_cast` (рассматриваемый в разделе 19.2.1). В качестве альтернативы, когда известно, что преобразование из базового в производный безопасно, для обхода запрета компилятора можно использовать оператор `static_cast` (см. раздел 4.11.3).



...и нет преобразований между объектами

Автоматическое преобразование производного класса в базовый применимо только для ссылок и указателей. Нет способа преобразования типа производного класса в тип базового класса. Однако нередко вполне возможно преобразовать объект производного класса в тип базового класса. Но такие преобразования не всегда ведут себя так, как хотелось бы.

Помните, что при инициализации или присвоении объекта типа класса фактически происходит вызов функции. При инициализации происходит вызов конструктора (см. раздел 13.1.1 и раздел 13.6.2), а при присвоении — вызов оператора присвоения (см. раздел 13.1.2 и раздел 13.6.2). У этих функций-членов обычно есть параметр, являющийся ссылкой на константную версию типа класса.

Поскольку эти функции-члены получают ссылки, преобразование производного класса в базовый позволяет передавать функциям копирования и перемещения базового класса объект производного класса. Эти функции не являются виртуальными. При передаче объекта производного класса конструктору базового выполняется конструктор, определенный в базовом классе. Этому конструктору известно *только* о членах самого базового класса. Точно так же, если объект производного класса присваивается объекту базового, выполняется оператор присвоения, определенный в базовом классе. Этот оператор также знает *только* о членах самого базового класса.

Например, классы приложения книжного магазина используют

синтезируемые версии операторов копирования и присвоения (см. раздел 13.1.1 и раздел 13.1.2). Более подробная информация об управлении копированием и наследовании приведена в разделе 15.7.2, а пока достаточно знать, что синтезируемые версии осуществляют почленное копирование или присвоение переменных-членов класса тем же способом, что и у любого другого класса:

```
Bulk_quote bulk; // объект производного типа
Quote item( bulk ); // используется конструктор
                      // Quote::Quote( const Quote& )
item = bulk;          // вызов Quote::operator=( const
Quote& )
```

При создании объекта `item` выполняется конструктор копий класса `Quote`. Этот конструктор знает только о переменных-членах `bookNo` и `price`. Он копирует эти члены из части `Quote` объекта `bulk` и игнорирует члены, являющиеся частью `Bulk_quote` объекта `bulk`. Аналогично при присвоении объекта `bulk` объекту `item` ему присваивается только часть `Quote` объекта `bulk`.

Поскольку часть `Bulk_quote` игнорируется, говорят, что она была *отсечена* (*sliced down*).



При инициализации объекта базового типа (или присвоении) объектом производного типа копируется, перемещается или присваивается только часть базового класса производного объекта. Производная часть объекта игнорируется.

Ключевая концепция. Преобразования между типами, связанными наследованием

Есть три правила преобразования связанных наследованием классов, о которых следует помнить.

- Преобразование из производного класса в базовый применимо только к указателю или ссылке.
- Нет неявного преобразования из типа базового класса в тип производного.
- При преобразовании производного в базовый член класса может быть недоступен из за спецификатора управления доступом. Доступность рассматривается в разделе 15.5.

Хотя автоматическое преобразование применимо только к указателям и ссылкам, большинство классов в иерархии наследования (явно или неявно) определяют функции-члены управления копированием (см. главу 13). В результате зачастую вполне можно копировать, перемещать и присваивать объекты производного типа объектам базового. Однако копирование, перемещение или присвоение объекта производного типа объекту базового копирует, перемещает или присваивает *только* члены части базового класса объекта.

Упражнения раздела 15.2.3

Упражнение 15.8. Определите статический и динамический типы.

Упражнение 15.9. Когда может возникнуть отличие статического типа выражения от его динамического типа? Приведите три примера, в которых статический и динамический типы отличаются.

Упражнение 15.10. Возвращаясь к обсуждению в разделе 8.1, объясните, как работает программа из раздела 8.2.1, где функции `read()` класса `Sales_data` передавался объект `ifstream`.



15.3. Виртуальные функции

Как уже упоминалось, в языке C++ динамическое связывание происходит при вызове виртуальной функции-члена через ссылку или указатель на тип базового класса (см. раздел 15.1). Поскольку до времени выполнения неизвестно, какая версия функции вызывается, виртуальные функции следует определять *всегда*. Обычно, если функция не используется, ее определение предоставлять необязательно (см. раздел 6.1.2). Однако следует определить каждую виртуальную функцию, независимо от того, будет ли она использована, поскольку у компилятора нет никакого способа определить, используется ли виртуальная функция.

Вызовы виртуальной функции могут быть распознаны во время выполнения

Когда виртуальная функция вызывается через ссылку или указатель, компилятор создает код *распознавания во время выполнения* (decide at run time) вызываемой функции. Вызывается та функция, которая соответствует динамическому типу объекта, связанного с этим указателем или ссылкой.

В качестве примера рассмотрим функцию `print_total()` из раздела 15.1. Она вызывает функцию `net_price()` своего параметра `item` типа `Quote&`. Поскольку параметр `item` — это ссылка и функция `net_price()` является виртуальной, какая именно из ее версий будет вызвана во время выполнения, зависит от фактического (динамического) типа аргумента, связанного с параметром `item`:

```
Quote base("0-201-82470-1", 50);
print_total(cout, base, 10); // вызов
Quote::net_price()
Bulk_quote derived("0-201-82470-1", 50, 5, .19);
print_total(cout, derived, 10); // вызов
Bulk_quote::net_price()
```

В первом вызове параметр `item` связан с объектом типа `Quote`. В результате, когда функция `print_total()` вызовет функцию `net_price()`, выполнится ее версия, определенная в классе `Quote`. Во втором вызове параметр `item` связан с объектом класса `Bulk_quote`. В этом вызове функция `print_total()` вызывает версию функции

`net_price()` класса `Bulk_quote`.

Крайне важно понимать, что динамическое связывание происходит только при вызове виртуальной функции через указатель или ссылку.

```
base = derived;           // копирует часть Quote  
производного в базовый  
base.net_price(20); // вызов Quote::net_price()
```

Когда происходит вызов виртуальной функции в выражении с обычным типом (не ссылкой и не указателем), такой вызов привязывается во время компиляции. Например, когда происходит вызов функции `net_price()` объекта `base`, нет никаких вопросов о выполняемой версии. Можно изменить значение (т.е. содержимое) объекта, который представляет `base`, но нет никакого способа изменить тип этого объекта. Следовательно, этот вызов распознается во время компиляции как версия `Quote::net_price()`.

Ключевая концепция. Полиморфизм в языке C++

Одной из ключевых концепций ООП является *полиморфизм* (polymorphism). В переводе с греческого языка "полиморфизм" означает множество форм. Связанные наследованием типы считаются полиморфными, поскольку вполне можно использовать многообразие форм этих типов, игнорируя различия между ними. Краеугольным камнем поддержки полиморфизма в языке C++ является тот факт, что статические и динамические типы ссылок и указателей могут отличаться.

Когда при помощи ссылки или указателя на базовый класс происходит вызов функции, определенной в базовом классе, точный тип объекта, для которого будет выполняться функция, неизвестен. Это может быть объект базового класса, а может быть и производного. Если вызываемая функция не виртуальная, независимо от фактического типа объекта, выполнена будет та версия функции, которая определена в базовом классе. Если функция виртуальная, решение о фактически выполняемой версии функции откладывается до времени выполнения. Она определяется на основании типа объекта, с которым связана ссылка или указатель.

С другой стороны, вызовы невиртуальных функций связываются во время компиляции. Точно так же вызовы любой функции (виртуальной или нет) для объекта связываются во время компиляции. Тип объекта фиксирован и неизменен — никак нельзя заставить динамический тип объекта отличаться от его статического типа. Поэтому вызовы для

объекта связываются во время компиляции с версией, определенной типом объекта.



Виртуальные функции распознаются во время выполнения, *только если* вызов осуществляется через ссылку или указатель. Только в этих случаях динамический тип объекта может отличаться от его статического типа.

Виртуальные функции в производном классе

При переопределении виртуальной функции производный класс может, но не обязан, повторить ключевое слово `virtual`. Как только функция объявляется виртуальной, она остается виртуальной во всех производных классах.

У функции производного класса, переопределяющей унаследованную виртуальную функцию, должны быть точно такие же типы параметров, как и у функции базового класса, которую она переопределяет.

За одним исключением тип возвращаемого значения виртуальной функции в производном классе также должен соответствовать типу возвращаемого значения функции в базовом классе. Исключение относится к виртуальным функциям, возвращающим ссылку (или указатель) на тип, который сам связан наследованием. Таким образом, если тип D происходит от типа B, то виртуальная функция базового класса может возвратить указатель на тип B*, а ее версия в производном классе может возвратить указатель на тип D*. Но такие типы возвращаемого значения требуют, чтобы преобразование производного класса в базовый из типа D в тип B было доступно. Доступность базового класса рассматривается в разделе 15.5. Пример такого вида виртуальной функции рассматривается в разделе 15.8.1.



Функция, являющаяся виртуальной в базовом классе, неявно остается виртуальной в его производных классах. Когда производный класс переопределяет виртуальную функцию, ее параметры в базовом и производных классах должны точно совпадать.

*Спецификаторы **final** и **override***

Как будет продемонстрировано в разделе 15.6, производный класс вполне может определить функцию с тем же именем, что и виртуальная функция в его базовом классе, но с другим списком параметров. Компилятор полагает, что такая функция независима от функции базового класса. В таких случаях версия в производном классе не переопределяет версию в базовом. На практике такие объявления зачастую являются ошибкой — автор класса намеревался переопределить виртуальную функцию базового класса, но сделал ошибку в определении списка параметров.



Поиск таких ошибок может быть на удивление трудным. По новому стандарту можно задать переопределение виртуальной функции в производном классе. Это дает ясно понять наше намерение и (что еще более важно) позволяет компилятору самому находить такие проблемы. Компилятор отвергнет программу, если функция, отмеченная как `override`, не переопределит существующую виртуальную функцию:

```
struct B {  
    virtual void f1(int) const;  
    virtual void f2();  
    void f3();  
};  
struct D1 : B {  
    void f1(int) const override; // ok: f1()  
    // соответствует f1() базового  
    void f2(int) override; // ошибка: B не имеет  
    // функции f2( int )  
    void f3() override; // ошибка: f3( ) не  
    // виртуальная функция  
    void f4() override; // ошибка: B не имеет  
    // функции f4( )  
};
```

В структуре `D1` спецификатор `override` для функции `f1()` вполне подходит; и базовые, и производные версии функции-члена `f1()` константы, они получают тип `int` и возвращают `void`. Версия `f1()` в структуре `D1` правильно переопределяет виртуальную функцию, которую

она унаследовала от структуры B.

Объявление функции f2() в структуре D1 не соответствует объявлению функции f2() в структуре B — она не получает никаких аргументов, а определенная в структуре D1 получает аргумент типа int. Поскольку объявления не совпадают, функция f2() в структуре D1 не переопределяет функцию f2() структуры B; это новая функция со случайно совпавшим именем. Как уже упоминалось, это объявление должно было быть переопределено, но этого не произошло и компилятор сообщил об ошибке.

Поскольку переопределена может быть только виртуальная функция, компилятор отвергнет также функцию f3() в структуре D1. Эта функция не виртуальна в структуре B, поэтому нечего и переопределять.

Точно так же ошибочна и функция f4(), поскольку в структуре B даже нет такой функции.

Функцию можно также определить как final. Любая попытка переопределения функции, которая была определена со спецификатором final, будет помечена как ошибка:

```
struct D2 : B {  
    // наследует f2() и f3() из B и переопределяет  
    f1(int)  
    void f1(int) const final; // последующие классы не  
    могут  
                                // переопределять  
f1(int)  
};  
struct D3 : D2 {  
    void f2(); // ok: переопределение f2()  
    // унаследованной от косвенно  
    // базовой структуры B  
    void f1(int) const; // ошибка: D2 объявила f2()  
    как final  
};
```

Спецификаторы final и override располагаются после списка параметров (включая квалификаторы ссылки или const) и после замыкающего типа (см. раздел 6.3.3).

Виртуальные функции и аргументы по умолчанию

Подобно любой другой функции, виртуальная функция может иметь

аргументы по умолчанию (см. раздел 6.5.1). Если вызов использует аргумент по умолчанию, то используемое значение определяется статическим типом, для которого вызвана функция.

Таким образом, при вызове через ссылку или указатель на базовый класс аргумент (аргументы) по умолчанию будет определен в базовом классе. Аргументы базового класса будут использоваться даже тогда, когда выполняется версия функции производного класса. В данном случае функции производного класса будут переданы аргументы по умолчанию, определенные для версии функции базового класса. Если функция производного класса будет полагаться на передачу других аргументов, то программа не будет выполняться, как ожидалось.

Рекомендуем

Виртуальные функции с аргументами по умолчанию должны использовать те же значения аргументов в базовом и производных классах.

Хитрость виртуального механизма

В некоторых случаях необходимо предотвратить динамическое связывание вызова виртуальной функции; нужно вынудить вызов использовать конкретную версию этой виртуальной функции. Для этого используется оператор области видимости. Рассмотрим, например, этот код:

```
// вызов версии базового класса независимо от  
динамического типа baseP
```

```
double undiscounted = baseP->Quote::net_price( 42 );
```

Здесь происходит вызов версии функции `net_price()` класса `Quote` независимо от типа объекта, на который фактически указывает `baseP`. Этот вызов будет распознан во время компиляции.



Обычно только код функций-членов (или друзей) должен использовать оператор области видимости для обхода виртуального механизма.

Зачем обходить виртуальный механизм? Наиболее распространен случай, когда виртуальная функция производного класса вызывает версию

базового класса. В таких случаях версия базового класса могла бы выполнять действия, общие для всей иерархии типов. Версии, определенные в производных классах, осуществляли бы любые дополнительные действия, специфичные для их собственного типа.



Если виртуальная функция производного класса, намереваясь вызвать свою версию из базового класса, пропустит оператор области видимости, то вызов будет распознан во время выполнения как вызов самой версии производного класса, что приведет к бесконечной рекурсии.

Упражнения раздела 15.3

Упражнение 15.11. Добавьте в иерархию класса `Quote` виртуальную функцию `debug()`, отображающую переменные-члены соответствующих классов.

Упражнение 15.12. Возможен ли случай, когда полезно объявить функцию-член и как `override`, и как `final`? Объясните, почему.

Упражнение 15.13. С учетом следующих классов объясните каждую из функций `print()`:

```
class base {
public:
    string name() { return basename; }
    virtual void print(ostream &os) { os << basename;
}
private:
    string basename;
};

class derived : public base {
public:
    void print(ostream &os) { print(os); os << " " <<
i; }
private:
    int i;
};
```

Если в этом коде имеются ошибки, устранитe их.

Упражнение 15.14. С учетом классов из предыдущего упражнения и следующих объектов укажите, какие из версий функций будут применены

во время выполнения:

```
base bobj;    base *bp1 = &bobj; base &br1 = bobj;
derived dobj; base *bp2 = &dobj; base &br2 = dobj;
( a)    bobj.print();   ( b)    dobj.print();   ( c)    bp1-
>name();
( d)    bp2->name();      ( e)    br1.print();      ( f)
br2.print();
```

15.4. Абстрактные базовые классы

Предположим, что классы приложения книжного магазина необходимо дополнить поддержкой нескольких стратегий скидок. Кроме оптовой скидки, можно было бы предоставить скидку за покупку до определенного количества, а свыше применять полную цену. Либо можно было бы предоставить скидку за покупку свыше одного предела, но не выше другого.

Для всех этих стратегий необходимы одинаковые средства: количество экземпляров и объем скидки. Для поддержки этих столь разных стратегий можно определить новый класс по имени `Disc_quote`, позволяющий хранить количество экземпляров и объем скидки. Такие классы как `Bulk_item`, предоставляющие определенную стратегию скидок, наследуются от класса `Disc_quote`. Каждый из производных классов реализует собственную стратегию скидок, определяя собственную версию функции `net_price()`.

Прежде чем определять собственный класс `Disc_quote`, следует решить, что будет делать функция `net_price()`. Класс `Disc_quote` не будет соответствовать никакой конкретной стратегии скидок; для этого класса нет никакого смысла создавать функцию `net_price()`.

Класс `Disc_quote` можно было бы определить без его собственной версии функции `net_price()`. В данном случае класс `Disc_quote` наследовал бы функцию `net_price()` от класса `Quote`.

Однако такой проект позволил бы пользователям писать бессмысленный код. Пользователь мог бы создать объект типа `Disc_quote`, предоставив количество и объем скидки. Передача объекта класса `Disc_quote` такой функции, как `print_total()`, задействовала бы версию функции `net_price()` из класса `Quote`. Вычисляемая цена не включила бы скидку, предоставляемую при создании объекта. Такое поведение не имеет никакого смысла.

Чистые виртуальные функции

Тщательный анализ этого вопроса показывает, что проблема не только в том, что неизвестно, как определить функцию `net_price()`. Практически следовало бы запретить пользователям создавать объекты класса `Disc_quote` вообще. Этот класс представляет общую концепцию скидки на книги, а не конкретную стратегию скидок.

Для воплощения этого намерения (и однозначного уведомления о бессмыслиности функции `net_price()`) определим функцию `net_price()` как *чистую виртуальную* функцию (*pure virtual*). В отличие от обычных виртуальных функций, чистая виртуальная функция не должна быть определена. Для определения виртуальной функции как чистой вместо ее тела используется часть = 0 (т.е. как раз перед точкой с запятой, завершающей объявление). Часть = 0 может присутствовать только в объявлении виртуальной функции в теле класса:

```
// класс для содержания объема скидки и количества
экземпляров
// используя эти данные, производные классы
реализуют стратегии скидок
class Disc_quote : public Quote {
public:
    Disc_quote() = default;
    Disc_quote(const std::string& book, double price,
               std::size_t qty, double disc):
        Quote(book, price), quantity(qty), discount(disc)
    { }
    double net_price(std::size_t) const = 0;
protected:
    std::size_t quantity = 0; // минимальная покупка
    для скидки
    double discount = 0.0; // доля применяемой
    скидки
};
```

Подобно прежнему классу `Bulk_item`, класс `Disc_quote` определяет стандартный конструктор и конструктор, получающий четыре параметра. Хотя объекты этого типа нельзя создавать непосредственно, конструкторы в классах, производных от класса `Disc_quote`, будут использовать конструкторы `Disc_quote()` для построения части `Disc_quote` своих объектов. Конструктор с четырьмя параметрами

передает первые два конструктору `Quote()`, а двумя последними непосредственно инициализирует собственные переменные-члены `discount` и `quantity`. Стандартный конструктор инициализирует эти члены значениями по умолчанию.

Следует заметить, что определение для чистой виртуальной функции предоставить нельзя. Однако тело функции следует определить вне класса. Поэтому нельзя предоставить в классе тело функции, для которой использована часть = 0.

Классы с чистыми виртуальными функциями являются абстрактными

Класс, содержащий (или унаследовавший без переопределения) чистую виртуальную функцию, является *абстрактным классом* (abstract base class). Абстрактный класс определяет интерфейс для переопределения последующими классами. Нельзя (непосредственно) создавать объекты абстрактного класса. Поскольку класс `Disc_quote` определяет функцию `net_price()` как чистую виртуальную, нельзя определить объекты типа `Disc_quote`. Можно определить объекты классов, производных от `Disc_quote`, если они переопределят функцию `net_price()`:

```
// Disc_quote объявляет чистые виртуальные функции,
которые
// переопределит Bulk_quote
Disc_quote discounted; // ошибка: нельзя определить
объект Disc_quote
Bulk_quote bulk; // ok: у Bulk_quote нет чистых
виртуальных функций
```

Классы, унаследованные от класса `Disc_quote`, должны определить функцию `net_price()`, иначе они также будут абстрактными.



Нельзя создать объекты абстрактного класса.

Конструктор производного класса инициализирует только свой прямой базовый класс

Теперь можно повторно реализовать класс `Bulk_quote` так, чтобы он происходил от класса `Disc_quote`, а не непосредственно от класса

Quote:

```
// скидка прекращается при продаже определенного количества экземпляров
// скидка выражается как доля сокращения полной цены
class Bulk_quote : public Disc_quote {
public:
    Bulk_quote() = default;
    Bulk_quote(const std::string& book, double price,
               std::size_t qty, double disc):
        Disc_quote(book, price, qty, disc) {}
    // переопределение базовой версии для реализации политики скидок
    double net_price(std::size_t) const override;
};
```

У этой версии класса Bulk_quote есть *прямой базовый класс* (direct base class), Disc_quote, и *косвенный базовый класс* (indirect base class), Quote. У каждого объекта класса Bulk_quote есть три внутренних объекта: часть Bulk_quote (пустая), часть Disc_quote и часть Quote.

Как уже упоминалось, каждый класс контролирует инициализацию объектов своего типа. Поэтому, даже при том, что у класса Bulk_quote нет собственных переменных-членов, он предоставляет тот же конструктор на четыре аргумента, что и первоначальный класс. Новый конструктор передает свои аргументы конструктору класса Disc_quote. Этот конструктор, в свою очередь, запускает конструктор Quote(). Конструктор Quote() инициализирует переменные-члены bookNo и price объекта bulk. Когда конструктор Quote() завершает работу, начинает работу конструктор Disc_quote(), инициализирующий переменные-члены quantity и discount. Теперь возобновляет работу конструктор Bulk_quote(). Он не делает ничего и ничего не инициализирует.

Ключевая концепция. Рефакторинг

Добавление класса Disc_quote в иерархию Quote является примером *рефакторинга* (refactoring). Рефакторинг подразумевает переделку иерархии классов с передачей некоторых функций и/или данных из одного класса в другой. Рефакторинг весьма распространен в объектно-ориентированных приложениях.

Примечательно, что, несмотря на изменение иерархии наследования, код, который использует классы `Bulk_quote` и `Quote`, изменять не придется. Но после рефакторинга классов (или любых других измененный) следует перекомпилировать весь код, который использует эти классы.

Упражнения раздела 15.4

Упражнение 15.15. Определите собственные версии классов `Disc_quote` и `Bulk_quote`.

Упражнение 15.16. Перепишите класс из упражнения 15.2.2 раздела 12.1.6, представляющий ограниченную стратегию скидок, так, чтобы он происходил от класса `Disc_quote`.

Упражнение 15.17. Попытайтесь определить объект типа `Disc_quote` и посмотрите, какие сообщения об ошибке выдал компилятор.



15.5. Управление доступом и наследование

Подобно тому, как каждый класс контролирует инициализацию своих переменных-членов (см. раздел 15.2.2), каждый класс контролирует также *доступность* (accessible) своих членов для производного класса.

Защищенные члены

Как уже упоминалось, класс использует защищенные члены в тех случаях, когда желает предоставить к ним доступ из производных классов, но защитить их от общего доступа. Спецификатор доступа `protected` можно считать гибридом спецификаторов `private` и `public`.

- Подобно закрытым, защищенные члены недоступны пользователям класса.
- Подобно открытым, защищенные члены доступны для членов и друзей классов, производных от данного класса.

Кроме того, защищенный член имеет еще одно важное свойство.

• Производный член класса или дружественный класс может обратиться к защищенным членам базового класса только *через* объект производного. У производного класса нет никакого специального способа доступа к защищенным членам объектов базового класса.

Чтобы лучше понять это последнее правило, рассмотрим следующий пример:

```
class Base {  
protected:  
    int prot_mem; // защищенный член  
};  
class Sneaky : public Base {  
    friend void clobber(Sneaky&); // есть доступ к  
Sneaky::prot_mem  
    friend void clobber(Base&); // нет доступа к  
Base::prot_mem  
    int j; // j по умолчанию  
закрытая  
};  
// ok: clobber может обращаться к закрытым и
```

зашитненным членам Sneaky

```
void clobber( Sneaky &s) { s.j = s.prot_mem = 0; }
// ошибка: clobber не может обращаться к защищенным
членам Base
```

```
void clobber( Base &b) { b.prot_mem = 0; }
```

Если производные классы (и друзья) смогут обращаться к защищенным членам в объекте базового класса, то вторая версия функции `clobber` (получающая тип `Base&`) будет корректна. Хоть эта функция и не дружественна классу `Base`, она все же сможет изменить объект типа `Base`; для обхода защиты спецификатором `protected` любого класса достаточно определить новый класс по линии `Sneaky`.

Для предотвращения такого способа применения члены и друзья производного класса могут обращаться к защищенным членам *только* тех объектов базового класса, которые встроены в объект производного; к обычным объектам базового типа у них никакого доступа нет.

Открытое, закрытое и защищенное наследование

Доступ к члену наследуемого класса контролируется комбинацией спецификатора доступа этого члена в базовом классе и спецификатором доступа в списке наследования производного класса. Для примера рассмотрим следующую иерархию:

```
class Base {
public:
    void pub_mem(); // открытый член
protected:
    int prot_mem; // защищенный член
private:
    char priv_mem; // закрытый член
};

struct Pub_Derv : public Base {
    // ok: производный класс имеет доступ к защищенным
членам
    int f() { return prot_mem; }
    // ошибка: закрытые члены недоступны производным
классам
    char g() { return priv_mem; }
};

struct Priv_Derv : private Base {
    // закрытое наследование не затрагивает доступ в
```

производном классе

```
int f1() const { return prot_mem; }  
};
```

Спецификатор доступа наследования никак не влияет на возможность членов (и друзей) производного класса обратиться к членам его собственного прямого базового класса. Доступ к членам базового класса контролируется спецификаторами доступа в самом базовом классе. Структуры `Pub_Derv` и `Priv_Derv` могут обращаться к защищенному члену `prot_mem`, но ни одна из них не может обратиться к закрытому члену `priv_mem`.

Задача спецификатора доступа наследования — контролировать доступ пользователей производного класса, включая другие классы, производные от него, к членам, унаследованным от класса `Base`:

```
Pub_Derv d1; // члены, унаследованные от Base,  
являются открытыми  
Priv_Derv d2; // члены, унаследованные от Base,  
являются закрытыми  
d1.pub_mem(); // ok: pub_mem является открытой в  
производном классе  
d2.pub_mem(); // ошибка: pub_mem является закрытой  
в производном классе
```

Структуры `Pub_Derv` и `Priv_Derv` унаследовали функцию `pub_mem()`. При открытом наследовании члены сохраняют свой спецификатор доступа. Таким образом, объект `d1` может вызвать функцию `pub_mem()`. В структуре `Priv_Derv` члены класса `Base` являются закрытыми; пользователи этого класса не смогут вызвать функцию `pub_mem()`.

Спецификатор доступа наследования, используемый производным классом, также контролирует доступ из классов, унаследованных от этого производного класса:

```
struct Derived_from_Public : public Pub_Derv {  
    // ok: Base::prot_mem остается защищенной в  
    Pub_Derv  
    int use_base() { return prot_mem; }  
};  
struct Derived_from_Private : public Priv_Derv {  
    // ошибка: Base::prot_mem является закрытой в  
    Priv_Derv
```

```
int use_base() { return prot_mem; }
};
```

Классы, производные от структуры `Pub_Derv`, могут обращаться к переменной-члену `prot_mem` класса `Base`, поскольку она остается защищенным членом в структуре `Pub_Derv`. У классов, производных от структуры `Priv_Derv`, напротив, такого доступа нет. Все члены, которые структура `Priv_Derv` унаследовала от класса `Base`, являются закрытыми.

Если бы был определен другой класс, скажем, `Prot_Derv`, использующий защищенное наследование, открытые члены класса `Base` в этом классе будут защищенными. У пользователей структуры `Prot_Derv` не было бы никакого доступа к функции `pub_mem()`, но ее члены и друзья могли бы обратиться к унаследованному члену.



Доступность преобразования производного класса в базовый класс

Будет ли доступно преобразование производного класса в базовый класс (см. раздел 15.2.2), зависит от того, какой код пытается использовать преобразование, а также от спецификатора доступа, используемого при наследовании производного класса. С учетом, что класс `D` происходит от класса `B`:

- Пользовательский код может использовать преобразование производного класса в базовый, *только если* класс `D` открыто наследует класс `B`. Пользовательский код не может использовать преобразование, если наследование было защищенным или закрытым.
- Функции-члены и друзья класса `D` могут использовать преобразование в `B` независимо от вида наследования `D` от `B`. Преобразование производного в прямой базовый класс всегда доступно для членов и друзей производного класса.
- Функции-члены и друзья классов, производных от класса `D`, могут использовать преобразование производного класса в базовый, если наследование было открытым или защищенным. Такой код не сможет использовать преобразование, если наследование классом `D` класса `B` было закрытым.



В любом месте кода, где доступен открытый член базового класса, будет доступно также преобразование производного класса в базовый, но не наоборот.

Ключевая концепция. Проект класса и защищенные члены

Без наследования у класса будет два разных вида пользователей: обычные пользователи и *разработчики* (*implementor*). Обычные пользователи пишут код, который использует объекты типа класса; такой код может обращаться только к открытым членам класса (интерфейсу). Разработчики пишут код, содержащийся в членах и друзьях класса. Члены и друзья класса могут обращаться и к открытым, и к закрытым разделам (реализации).

При наследовании появляется третий вид пользователей, а именно производные классы. Базовый класс делает защищенными те части своей реализации, которые позволено использовать его производным классам. Защищенные члены остаются недоступными обычному пользовательскому коду; закрытые члены остаются недоступными производным классам и их друзьям.

Подобно любому другому классу, базовый класс объявляет члены своего интерфейса открытыми. Класс, используемый как базовый, может разделить свою реализацию на члены, доступные для производных классов и доступные только для базового класса и его друзей. Член класса, относящийся к реализации, должен быть защищен, если он предоставляет функцию или данные, которые производный класс должен будет использовать в собственной реализации. В противном случае члены реализации должны быть закрытыми.

Дружественные отношения и наследование

Подобно тому, как дружественные отношения не передаются (см. раздел 7.3.4), они также не наследуются. У друзей базового класса нет никаких специальных прав доступа к членам его производных классов, а у друзей производного класса нет специальных прав доступа к базовому классу:

```
class Base {  
    // добавлено объявление; другие члены, как прежде  
    friend class Pal; // у Pal нет доступа к классам,  
    производным от Base  
};
```

```

class Pal {
public:
    int f( Base b) { return b.prot_mem; } // ok: Pal
дружествен Base
    int f2( Sneaky s) { return s.j; } // ошибка:
Pal не
                                            // дружествен
Sneaky
    // доступ к базовому классу контролируется базовым
классом, даже в
    // объекте производного
    int f3( Sneaky s) { return s.prot_mem; } // ok: Pal
дружествен
} ;

```

Факт допустимости функции `f3()` может показаться удивительным, но он непосредственно следует из правила, что все классы контролируют доступ к собственным членам. Класс `Pal` — друг класса `Base`, поэтому класс `Pal` может обращаться к членам объектов класса `Base`. Это относится и к встроенным в объект класса `Base` объектам классов, производных от него.

Когда класс объявляет другой класс дружественным, это относится только к данному классу, ни его базовые, ни производные классы никаких специальных прав доступа не имеют:

```

// у D2 нет доступа к закрытым или защищенным
членам Base
class D2 : public Pal {
public:
    int mem( Base b)
    { return b.prot_mem; } // ошибка: дружба не
наследуется
} ;

```



Дружественные отношения не наследуются; каждый класс сам контролирует доступ к своим членам.

Освобождение индивидуальных членов

Иногда необходимо изменить уровень доступа к имени, унаследованному производным классом. Для этого можно использовать объявление `using` (см. раздел 3.1):

```
class Base {  
public:  
    std::size_t size() const { return n; }  
protected:  
    std::size_t n;  
};  
class Derived : private Base { // заметьте,  
наследование закрытое  
public:  
    // обеспечить уровня доступа для членов, связанных  
    с размером объекта  
    using Base::size;  
protected:  
    using Base::n;  
};
```

Поскольку класс `Derived` использует закрытое наследование, унаследованные члены `size()` и `n` по умолчанию будут закрытыми членами класса `Derived`. Объявления `using` корректируют доступность этих членов. Пользователи класса `Derived` могут обращаться к функции-члену `size()`, а классы, впоследствии произошедшие от класса `Derived`, смогут обратиться к переменной `n`.

Объявление `using` в классе может использовать имя любого доступного (не закрытого) члена прямого или косвенного базового класса. Доступность имени, указанного в объявлении `using`, зависит от спецификатора доступа, предшествующего объявлению `using`. Таким образом, если объявление `using` расположено в разделе `private` класса, то имя будет доступно только для членов и друзей. Если объявление находится в разделе `public`, имя доступно для всех пользователей класса. Если объявление находится в разделе `protected`, имя доступно только для членов, друзей и производных классов.



Производный класс может предоставить объявление `using` только для тех

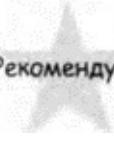
имен, доступ к которым разрешен.

Уровни защиты наследования по умолчанию

В разделе 7.2 упоминалось о том, что у классов, определенных с использованием ключевых слов `struct`, и `class` разные спецификаторы доступа по умолчанию. Точно так же заданный по умолчанию спецификатор наследования зависит от ключевого слова, используемого при определении производного класса. По умолчанию у производного класса, определенного с ключевым словом `class`, будет закрытое наследование (`private inheritance`), а с ключевым словом `struct` — открытое (`public inheritance`):

```
class Base { /* ... */ };
struct D1 : Base { /* ... */ }; // открытое
наследование по умолчанию
class D2 : Base { /* ... */ }; // закрытое
наследование по умолчанию
```

Весьма распространено заблуждение, что между классами и структурами есть иные, более глубокие различия. Единственное различие — заданные по умолчанию спецификаторы доступа для членов и наследования. Никаких других различий нет.

 Рекомендуем

Для закрытого наследования производный класс должен быть явно определен как `private`, не следует полагаться на поведение по умолчанию. Это ясно дает понять, что закрытое наследование применено преднамеренно, а не по оплошности.

Упражнения раздела 15.5

Упражнение 15.18. С учетом классов `Base` и производных от него, и типов объектов, приведенных в комментариях, укажите, какие из следующих присвоений допустимы. Объясните, почему некорректны недопустимые.

```
Base *p = &d1; // d1 имеет тип Pub_Derv
p = &d2;         // d2 имеет тип Priv_Derv
p = &d3;         // d3 имеет тип Prot_Derv
p = &dd1;        // dd1 имеет тип Derived_from_Public
p    =  &dd2;      // dd2 имеет тип
```

Derived_from_Private
p = &dd3; // dd3 имеет тип
Derived_from_Protected

Упражнение 15.19. Предположим, у каждого из классов: Base и производных от него, есть функция-член в формате

```
void memfn( Base &b) { b = *this; }
```

Укажите, была ли эта функция допустима для каждого класса.

Упражнение 15.20. Напишите код проверки ответов на предыдущие два упражнения.

Упражнение 15.21. Выберите одну из следующих общих абстракций, содержащих семейство типов (или любую собственную). Организуйте типы в иерархию наследования.

- (a) Форматы графических файлов (например: gif, tiff, jpeg, bmp)
- (b) Геометрические примитивы (например: box, circle, sphere, cone)
- (c) Типы языка C++ (например: class, function, member function)

Упражнение 15.22. Укажите имена некоторых из наиболее вероятных виртуальных функций, а также открытых и защищенных членов для класса, выбранного в предыдущем упражнении.



15.6. Область видимости класса при наследовании

Каждый класс определяет собственную *область видимости* (scope) (см. раздел 7.4), в рамках которой определены его члены. При наследовании область видимости производного класса (см. раздел 2.2.4) вкладывается в области видимости его базовых классов. Если имя не найдено в области видимости производного класса, поиск его определения продолжается в областях видимости базовых классов.

Тот факт, что область видимости производного класса вложена в область видимости его базовых классов, может быть удивителен. В конце концов, базовые и производные классы определяются в разных частях текста программы. Но именно это иерархическое вложение областей видимости класса позволяет членам производного класса использовать члены его базового класса, как будто они являются частью производного класса. Рассмотрим пример:

```
Bulk_quote bulk;
cout << bulk.isbn();
```

В этом коде поиск определения имени `isbn()` осуществляется следующим образом.

- Поскольку вызывается функция `isbn()` объекта типа `Bulk_quote`, поиск начинается в классе `Bulk_quote`. В этом классе имя `isbn()` не найдено.
- Поскольку класс `Bulk_quote` происходит от класса `Disc_quote`, в нем и продолжается поиск. Имя все еще не найдено.
- Поскольку класс `Disc_quote` происходит от класса `Quote`, поиск продолжается в нем. В этом классе находится определение имени `isbn()`; таким образом, вызов `isbn()` распознается как вызов функции `isbn()` класса `Quote`.

Поиск имен осуществляется во время компиляции

Статический тип (см. раздел 15.2.3) объекта, ссылки или указателя определяет, какие члены этого объекта будут видимы. Даже когда статический и динамический типы отличаются (это бывает в случае, когда используется ссылка или указатель на базовый класс), именно статический тип определяет применимые члены. Например, в класс `Disc_quote`

можно было бы добавить функцию-член, которая возвращает пару (тип *pair*) (см. раздел 11.2.3), содержащую минимальное (или максимальное) количество и цену со скидкой.

```
class Disc_quote : public Quote {  
public:  
    std::pair<size_t, double> discount_policy() const  
    { return {quantity, discount}; }  
    // другие члены как прежде  
};
```

Функцию *discount_policy()* можно использовать только через объект, указатель, или ссылку на тип *Disc_quote*, или класс, производный от него:

```
Bulk_quote bulk;  
Bulk_quote *bulkP = &bulk; // статический и  
динамический типы совпадают  
Quote *itemP = &bulk; // статический и динамический  
типы отличаются  
bulkP->discount_policy(); // ok: bulkP имеет тип  
Bulk_quote*  
itemP->discount_policy(); // ошибка: itemP имеет  
типа Quote*
```

Хотя объект *bulk* имеет функцию-член *discount_policy()*, она недоступна через указатель *itemP*. Тип *itemP* — указатель на тип *Quote*, а это значит, что поиск имени *discount_policy()* начнется в классе *Quote*. У класса *Quote* нет члена по имени *discount_policy()*, поэтому вызов этой функции-члена объекта, ссылки или указателя на тип *Quote* невозможен.

Конфликт имен и наследование

Как и любая другая, область видимости производного класса позволяет повторно использовать имя, определенное в его прямом или косвенном базовом классе. Как обычно, имена, определенные во внутренней области видимости (например, в производном классе), скрывают имена во внешней области видимости (например, в базовом классе) (см. раздел 2.2.4):

```
struct Base {  
    Base() : mem(0) {}  
protected:  
    int mem;
```

```

};

struct Derived : Base {
    Derived( int i) : mem( i) { } // Derived::mem
инициализируется i
                                            // Base::mem
инициализируется по умолчанию
    int get_mem( ) { return mem; } // возвращает
Derived::mem
protected:
    int mem; // скрывает mem в Base
};

```

Ссылка на переменную `mem` в функции `get_mem()` распознается как имя в классе `Derived`. Таким образом, код

```

Derived d( 42);
cout << d.get_mem( ) << endl; // выводит 42
выведет значение 42.

```



Член производного класса, имя которого совпадает с именем члена базового класса, скрывает член базового класса и предотвращает прямой доступ к нему.

Применение оператора области видимости для доступа к скрытым членам

Для доступа к скрытому члену базового класса можно использовать оператор области видимости.

```

struct Derived : Base {
    int get_base_mem( ) { return Base::mem; }
};

```

Оператор области видимости изменяет нормальный порядок поиска и заставляет компилятор начинать поиск имени `mem` с класса `Base`. Если бы код выше был выполнен с этой версией класса `Derived`, то результатом вызова `d. get_mem()` был бы 0.

 Рекомендуем

Кроме переопределения унаследованных виртуальных функций, производный класс обычно не должен повторно использовать имена, определенные в его базовом классе.

Ключевая концепция. Поиск имени и наследование

Для понимания наследования в языке C++ крайне важно знать, как распознаются вызовы функций. Процесс распознавания вызова `p->mem()` (или `obj. mem()`) проходит в четыре этапа.

- Сначала определяется статический тип объекта `p` (или `obj`). Поскольку это вызов члена класса, тип будет классом.

- Поиск имени `mem` осуществляется в классе, который соответствует статическому типу объекта `p` (или `obj`). Если функция `mem()` не найдена, поиск продолжается в прямом базовом классе и далее по цепи классов, пока имя `mem` не будет найдено или пока не будет осмотрен последний класс. Если функция `mem()` не будет найдена ни в самом классе, ни в его базовых классах, вызов откомпилирован не будет.

- Как только имя `mem` будет найдено, осуществляется обычная проверка соответствия типов (см. раздел 6.1), гарантирующая допустимость найденного определения для данного вызова.

- Если вызов допустим, компилятор создает код, зависящий от того, является ли вызываемая функция виртуальной или нет:

- Если функция `mem()` виртуальная и вызов осуществляется через ссылку или указатель, то компилятор создает код, который во время выполнения определяет на основании динамического типа объекта выполняемую версию функции.

- В противном случае, если функция не является виртуальной или если вызов осуществляется для объекта (а не ссылки или указателя), то компилятор создает код обычного вызова функции.

Как обычно, поиск имени осуществляется перед проверкой соответствия типов

Как уже упоминалось, функции, объявленные во внутренней области видимости, не перегружают функции, объявленные во внешней области видимости (см. раздел 6.4.1). В результате функции, определенные в производном классе, не перегружают функции-члены, определенные в его базовом классе (классах). Подобно любой другой области видимости, если имя члена производного класса (т.е. определенное во внутренней области видимости) совпадает с именем члена базового класса (т.е. именем во

внешней области видимости), то в рамках производного класса имя, определенное в производном классе, скрывает имя в базовом классе. Имя функции-члена базового класса скрывается, даже если у функций будут разные списки параметров:

```
struct Base {  
    int memfcn();  
};  
struct Derived : Base {  
    int memfcn(int); // скрывает memfcn() в базовом  
классе  
};  
Derived d; Base b;  
b.memfcn(); // вызов Base::memfcn()  
d.memfcn(10); // вызов Derived::memfcn()  
d.memfcn(); // ошибка: memfcn() без  
аргументов скрывается  
d.Base::memfcn(); // ok: вызов Base::memfcn()
```

Объявление функции `memfcn()` в классе `Derived` скрывает объявление функции `memfcn()` в классе `Base`. Не удивительно, что первый вызов через объект `b` класса `Base` вызывает версию в базовом классе. Точно так же второй вызов (через объект `d`) вызывает версию класса `Derived`. Удивительно то, что третий вызов, `d.memfcn()`, некорректен.

Чтобы распознать этот вызов, компилятор ищет имя `memfcn` в классе `Derived`. Этот класс определяет член по имени `memfcn`, и поиск на этом останавливается. Как только имя будет найдено, компилятор далее не ищет. Версия функции `memfcn()` в классе `Derived` ожидает аргумент типа `int`. Поскольку данный вызов такого аргумента не предоставляет, вызов ошибочен.



Виртуальные функции и область видимости

Теперь можно разобраться, почему у виртуальных функций должен быть одинаковый список параметров в базовом и производном классах (см. раздел 15.3). Если функции-члены в базовом и производном классах будут получать разные аргументы, не будет никакого способа вызвать версию производного класса через ссылку или указатель на базовый. Например:

```

class Base {
public:
    virtual int fcn();
};

class D1 : public Base {
public:
    // скрывает fcn() в базовом; функция fcn() не
    // виртуальна
    // D1 наследует определение из Base::fcn()
    int fcn( int );           // список параметров fcn() в
Base другой
    virtual void f2();        // новая виртуальная функция,
                            // не существующая в Base
};

class D2 : public D1 {
public:
    int fcn( int );          // невиртуальная функция скрывает
D1::fcn( int )
    int fcn();               // переопределяет виртуальную
функцию fcn() из Base
    void f2();               // переопределяет виртуальную
функцию f2() из D1
};

```

Функция `fcn()` в классе `D1` не переопределяет виртуальную функцию `fcn()` из класса `Base`, поскольку у них разные списки параметров. Вместо этого она *скрывает* функцию `fcn()` из базового класса. Фактически у класса `D1` есть две функции по имени `fcn()`: класс `D1` унаследовал виртуальную функцию `fcn()` от класса `Base`, а также определяет собственную невиртуальную функцию-член по имени `fcn()`, получающую параметр типа `int`.

Вызов скрытой виртуальной функции через базовый класс

С учетом классов, описанных выше, рассмотрим несколько разных способов вызова этих функций:

```

Base bobj; D1 d1obj; D2 d2obj;
Base *bp1 = &bobj, *bp2 = &d1obj, *bp3 = &d2obj;
bp1->fcn(); // виртуальный вызов Base::fcn() во
время выполнения

```

```

bp2->fcn( ); // виртуальный вызов Base::fcn( ) во время выполнения
bp3->fcn( ); // виртуальный вызов D2::fcn( ) во время выполнения
D1 *d1p = &d1obj; D2 *d2p = &d2obj;
bp2->f2(); // ошибка: Base не имеет члена по имени f2()
d1p->f2(); // виртуальный вызов D1::f2() во время выполнения
d2p->f2(); // виртуальный вызов D2::f2() во время выполнения

```

Все три первых вызова сделаны через указатели на базовый класс. Поскольку функция `fcn()` является виртуальной, компилятор создает код, способный во время выполнения решить, какую версию вызвать.

Это решение будет принято на основании фактического типа объекта, с которым связан указатель. В случае указателя `bp2` основной объект имеет тип `D1`. Этот класс не переопределит функцию `fcn()` без параметров. Таким образом, вызов через указатель `bp2` распознается (во время выполнения) как версия, определенная в классе `Base`.

Следующие три вызова осуществляются через указатели с различными типами. Каждый указатель указывает на один из типов в этой иерархии. Первый вызов некорректен, так как в классе `Base` нет функции `f2()`. Тот факт, что указатель случайно указывает на производный объект, является несущественным.

И наконец, рассмотрим вызовы невиртуальной функции `fcn(int)`:

```

Base *p1 = &d2obj; D1 *p2 = &d2obj; D2 *p3 =
&d2obj;
p1->fcn( 42 ); // ошибка: Base не имеет версии fcn( ), получающей int
p2->fcn( 42 ); // статическое связывание, вызов D1::fcn( int )
p3->fcn( 42 ); // статическое связывание, вызов D2::fcn( int )

```

В каждом вызове указатель случайно указывает на объект типа `D2`. Но динамический тип не имеет значения, когда происходит вызов невиртуальной функции. Вызываемая версия зависит только от статического типа указателя.

Переопределение перегруженных функций

Подобно любой другой функции, функция-член (виртуальная или нет) может быть перегружена. Производный класс способен переопределить любое количество экземпляров перегруженных функций, которые он унаследовал. Если производный класс желает сделать все перегруженные версии доступными через свой тип, то он должен переопределить их все или ни одну из них.

Иногда класс должен переопределить некоторые, но не все функции в наборе перегруженных. В таких случаях было бы весьма утомительно переопределять каждую версию базового класса, чтобы переопределить только те, которые должен специализировать класс.

Вместо переопределения каждой версии базового класса, которую он унаследовал, производный класс может предоставить объявление `using` (см. раздел 15.5) для перегруженного члена. Объявление `using` определяет только имя; оно не может определить список параметров. Таким образом, объявление `using` для функции-члена базового класса добавляет все перегруженные экземпляры этой функции в область видимости производного класса. Перенеся все имена в свою область видимости, производный класс должен определить только те функции, которые действительно зависят от его типа.

Обычные правила объявления `using` в классе относятся и к именам перегруженных функций (см. раздел 15.5); каждый перегруженный экземпляр функции в базовом классе должен быть доступен в производном классе. Доступ к перегруженным версиям, которые в противном случае не переопределяются производным классом, будет возможен в точке объявления `using`.

Упражнения раздела 15.6

Упражнение 15.23. Предположим, что класс `D1` намеревается переопределить свою унаследованную функцию `fcn()`. Как исправить этот класс? Предположим, что класс исправлен так, что функция `fcn()` соответствует определению в классе `Base`. Как бы распознавались вызовы в этом разделе?

15.7. Конструкторы и функции управления копированием

Подобно любому другому классу, класс в иерархии наследования контролирует происходящее при создании, копировании, перемещении, присвоении или удалении объектов его типа. Как и у любого другого класса, если класс (базовый или производный) сам не определяет одну из функций управления копированием, ее синтезирует компилятор. Кроме того, как обычно, синтезируемая версия любой из этих функций-членов может быть удаленной функцией.



15.7.1. Виртуальные деструкторы

Основное воздействие, которое наследование оказывает на управление копированием для базового класса, заключается в том, что базовый класс обычно должен определять виртуальный деструктор (см. раздел 15.2.1). Деструктор должен быть виртуальной функцией, чтобы обеспечить объектам в иерархии наследования возможность динамического создания.

Помните, что деструктор выполняется при удалении указателя на динамически созданный объект (см. раздел 13.1.3). Если это указатель на тип в иерархии наследования, вполне возможно, что статический тип указателя может отличаться от динамического типа удаляемого объекта (см. раздел 15.2.2). Например, при удалении указателя типа `Quote*` может оказаться, что он указывал на объект класса `Bulk_quote`. Если он указывает на объект типа `Bulk_quote`, компилятор должен знать, что следует выполнить деструктор именно класса `Bulk_quote`. Подобно любой другой функции, чтобы был выполнен надлежащий деструктор, в базовом классе его следует определить как виртуальную функцию:

```
class Quote {
public:
    // виртуальный деструктор необходим при удалении
    // указателя на
    // базовый тип, указывающего на объект
    // производного
    virtual ~Quote() = default; // динамическое
```

```
связывание для                                // деструктора  
};
```

Подобно любой другой виртуальной функции, виртуальный характер деструктора наследуется. Таким образом, у классов, производных от класса `Quote`, окажутся виртуальные деструкторы, будь то синтезируемый деструктор или собственный. Пока деструктор базового класса остается виртуальной функцией, при удалении указателя на базовый класс будет выполнен соответствующий деструктор:

```
Quote *itemP = new Quote; // статический и  
динамический типы совпадают  
delete itemP;           // вызов деструктора для  
Quote  
itemP = new Bulk_quote;  // статический и  
динамический типы разные  
delete itemP;           // вызов деструктора для  
Bulk_quote
```



Выполнение оператора `delete` для указателя на базовый класс, который указывает на объект производного класса, приведет к непредсказуемым последствиям, если деструктор базового класса не будет виртуальным.

Деструкторы базовых классов — важное исключение из эмпирических правил, согласно которым, если класс нуждается в деструкторе, то он также нуждается в функциях копирования и присвоения (см. раздел 13.6). Базовый класс почти всегда нуждается в деструкторе, поэтому он может сделать деструктор виртуальным. Если базовый класс обладает пустым деструктором, только чтобы сделать его виртуальным, то наличие у класса деструктора вовсе не означает, что также необходим оператор присвоения или конструктор копий.

Виртуальный деструктор отменяет синтез функций перемещения

Тот факт, что базовый класс нуждается в виртуальном деструкторе, имеет важное косвенное последствие для определения базовых и производных классов: если класс определит деструктор (даже с

использованием синтаксиса `= default`, чтобы использовать синтезируемую версию), то компилятор не будет синтезировать функцию перемещения для этого класса (см. раздел 13.6.2).

Упражнения раздела 15.7.1

Упражнение 15.24. Какие виды классов нуждаются в виртуальном деструкторе? Какие задачи должен выполнять виртуальный деструктор?



15.7.2. Синтезируемые функции управления копированием и наследование

Синтезируемые функции-члены управления копированием в базовом или производном классе выполняются, как любой другой синтезируемый конструктор, оператор присвоения или деструктор: они почленно инициализируют, присваивают или удаляют члены самого класса. Кроме того, эти синтезируемые члены инициализируют, присваивают или удаляют прямую базовую часть объекта при помощи соответствующей функции базового класса. Соответствующие примеры приведены ниже.

- Синтезируемый стандартный конструктор класса `Bulk_quote` запускает стандартный конструктор класса `Disc_quote`, который в свою очередь запускает стандартный конструктор класса `Quote`.

- Стандартный конструктор класса `Quote` инициализирует по умолчанию переменную-член `bookNo` пустой строкой и использует внутриклассовый инициализатор для инициализации переменной-члена `price` нулем.

- Когда конструктор класса `Quote` завершает работу, конструктор класса `Disc_quote` продолжает ее, используя внутриклассовые инициализаторы для инициализации переменных `qty` и `discount`.

- Когда завершает работу конструктор класса `Disc_quote`, конструктор класса `Bulk_quote` продолжает ее, но не выполняет никаких других действий.

Точно так же синтезируемый конструктор копий класса `Bulk_quote` использует (синтезируемый) конструктор копий класса `Disc_quote`, который использует (синтезируемый) конструктор копий класса `Quote`. Конструктор копий класса `Quote` копирует переменные-члены `bookNo` и `price`; а конструктор копий класса `Disc_quote` копирует переменные-

члены `qty` и `discount`.

Следует заметить, что не имеет значения, синтезируется ли функция-член базового класса (как в случае иерархии `Quote`) или имеет предоставленное пользователем определение. Важно лишь то, что соответствующая функция-член доступна (см. раздел 15.5) и что она не удаленная.

Каждый из классов иерархии `Quote` использует синтезируемый деструктор. Производные классы делают это неявно, тогда как класс `Quote` делает это явно, определяя свой (виртуальный) деструктор как `= default`. Синтезируемый деструктор (как обычно) пуст, и его неявная часть удаляет члены класса (см. раздел 13.1.3). В дополнение к удалению собственных членов фаза удаления деструктора в производном классе удаляет также свою прямую базовую часть. Этот деструктор в свою очередь вызывает деструктор своего прямого базового класса, если он есть. И так далее до корневого класса иерархии.

Как уже упоминалось, у класса `Quote` нет синтезируемых функций перемещения, поскольку он определяет деструктор. При каждом перемещении объекта `Quote` (см. раздел 13.6.2) будут использоваться (синтезируемые) функции копирования. Как будет продемонстрировано ниже, тот факт, что у класса `Quote` нет функций перемещения, означает, что его производные классы также не будут их иметь.



Базовые классы и удаленные функции управления копированием в производном классе

Синтезируемый стандартный конструктор или любая из функций-членов управления копированием базового либо производного класса может быть определена как удаленная по тем же причинам, что и в любом другом классе (см. раздел 13.1.6 и раздел 13.6.2). Кроме того, способ определения базового класса может вынудить член производного класса стать удаленным.

- Если стандартный конструктор, конструктор копий, оператор присвоения копии или деструктор в базовом классе удалены или недоступны (раздел 15.5), то соответствующая функция-член в производном классе определяется как удаленная, поскольку компилятор не может использовать функцию-член базового класса для создания, присвоения или удаления части объекта базового класса.

- Если у базового класса недоступен или удален деструктор, то синтезируемые стандартный конструктор и конструктор копий в производных классах определяются как удаленные, поскольку нет никакого способа удалить базовую часть производного объекта.

- Как обычно, компилятор не будет синтезировать удаленную функцию перемещения. Если использовать синтаксис = default для создания функции перемещения, то это будет удаленная функция в производном классе, если соответствующая функция в базовом классе будет удалена или недоступна, поскольку часть базового класса не может быть перемещена. Конструктор перемещения также будет удален, если деструктор базового класса окажется удален или недоступен.

Для примера рассмотрим базовый класс B:

```
class B {  
public:  
    B();  
    B(const B&) = delete;  
    // другие члены, исключая конструктор перемещения  
};  
class D : public B {  
    // нет конструкторов  
};  
D d; // ok: синтезируемый стандартный конструктор  
класса D использует  
    // стандартный конструктор класса B  
D d2(d); // ошибка: синтезируемый конструктор копий  
класса D удален  
D     d3(std::move(d)); // ошибка: неявно  
использованный удаленный  
    // конструктор копий класса D
```

Класс имеет доступный стандартный конструктор и явно удаленный конструктор копий. Поскольку конструктор копий определяется, компилятор не будет синтезировать для класса B конструктор перемещения (см. раздел 13.6.2). В результате невозможно ни переместить, ни скопировать объекты типа B. Если бы класс, производный от типа B, хотел позволить своим объектам копирование или перемещение, то этот производный класс должен был бы определить свои собственные версии этих конструкторов. Конечно, этот класс должен был бы решить, как скопировать или переместить члены в эту часть базового класса.

Практически, если у базового класса нет стандартного конструктора копий или конструктора перемещения, то его производные классы также обычно не будут их иметь.

Функции перемещения и наследование

Как уже упоминалось, большинство базовых классов определяет виртуальный деструктор. В результате по умолчанию базовые классы вообще не получают синтезируемых функций перемещения. Кроме того, по умолчанию классы, производные от базового класса, у которого нет функций перемещения, также не получают синтезируемых функций перемещения.

Поскольку отсутствие функции перемещения в базовом классе подавляет синтез функций перемещения в его производных классах, базовые классы обычно должны определять функции перемещения, если это имеет смысл. Наш класс `Quote` может использовать синтезируемые версии. Однако класс `Quote` должен определить эти члены явно. Как только он определит собственные функции перемещения, он должен будет также явно определить версии копирования (см. раздел 13.6.2):

```
class Quote {  
public:  
    Quote() = default; // почленная инициализация по  
// умолчанию  
    Quote( const Quote&) = default; // почленное  
// копирование  
    Quote( Quote&&) = default; // почленное  
// копирование  
    Quote& operator=( const Quote&) = default; //  
// присвоение копии  
    Quote& operator=( Quotes&) = default; //  
// перемещение  
    virtual ~Quote() = default;  
    // другие члены, как прежде  
};
```

Теперь объекты класса `Quote` будут почленно копироваться, перемещаться, присваиваться и удаляться. Кроме того, классы, производные от класса `Quote`, также автоматически получат синтезируемые функции перемещения, если у них не будет членов, которые воспрепятствуют перемещению.

Упражнения раздела 15.7.2

Упражнение 15.25. Зачем определять стандартный конструктор для класса `Disc_quote`? Как повлияет на поведение класса `Bulk_quote`, если вообще повлияет, удаление этого конструктора?



15.7.3. Функции-члены управления копированием производного класса

Как упоминалось в разделе 15.2.2, фаза инициализации конструктора производного класса инициализирует часть (части) базового класса производного объекта наряду с инициализацией его собственных членов. В результате конструкторы копирования и перемещения для производного класса должны копировать и перемещать члены своей базовой части наравне с производной. Точно так же оператор присвоения производного класса должен присваивать члены базовой части производного объекта.

В отличие от конструкторов и операторов присвоения, деструктор несет ответственность только за освобождение ресурсов, зарезервированных производным классом. Помните, что члены объекта освобождаются неявно (см. раздел 13.1.3). Точно так же часть базового класса объекта производного класса освобождается автоматически.



ВНИМАНИЕ

Когда производный класс определяет функцию копирования или перемещения, эта функция несет ответственность за копирование или перемещение всего объекта, включая члены базового класса.



Определение конструктора копии или перемещения производного класса

При определении конструктора копии или перемещения (см. раздел 13.1.1 и раздел 13.6.2) для производного класса обычно используется соответствующий конструктор базового класса, инициализирующий базовую часть объекта:

```
class Base { /* ... */ ;
```

```

class D: public Base {
public:
    // по умолчанию стандартный конструктор базового
    // класса
    // инициализирует базовую часть объекта
    // чтобы использовать конструктор копии или
    // перемещения, его следует
    // вызвать явно
    // конструктор в списке инициализации конструктора
    D( const D& d) : Base( d) // копирование базовых
    // членов
    /* инициализаторы для членов класса D */ { /* ... */
*/ }

    D( D&& d): Base( std::move( d)) // перемещение
    // базовых членов
    /* инициализаторы для членов класса D */ { /* ... */
*/ }
};


```

Инициализатор `Base(d)` передает объект класса `D` конструктору базового класса. Хотя в принципе у класса `Base` может быть конструктор с параметром типа `D`, на практике это очень маловероятно. Вместо этого инициализатор `Base(d)` будет (обычно) соответствовать конструктору копий класса `Base`. В этом конструкторе объект `d` будет связан с параметром типа `Base&`. Конструктор копий класса `Base` скопирует базовую часть объекта `d` в создаваемый объект. Будь инициализатор для базового класса пропущен, для инициализации базовой части объекта класса `D` будет использован стандартный конструктор класса `Base`.

```

    // вероятно, неправильное определение конструктора
    // копий D
    // часть базового класса инициализируется по
    // умолчанию, а не копией
    D( const D& d) /* инициализаторы членов класса, но
    // не базового класса */
    { /* ... */ }

```

Предположим, что конструктор класса `D` копирует производные члены объекта `d`. Этот вновь созданный объект был бы настроен странно: его члены класса `Base` содержали бы значения по умолчанию, в то время как его члены класса `D` были бы копиями данных из другого объекта.



ВНИМАНИЕ

По умолчанию стандартный конструктор базового класса инициализирует часть базового класса объекта производного. Если необходимо копирование (или перемещение) части базового класса, следует явно использовать конструктор копий (или перемещения) для базового класса в списке инициализации конструктора производного.

Оператор присвоения производного класса

Подобно конструктору копирования и перемещения, оператор присвоения производного класса (см. раздел 13.1.2 и раздел 13.6.2) должен присваивать свою базовую часть явно:

```
// Base::operator=(const Base&) не вызывается
автоматически
D &D::operator=(const D &rhs) {
    Base::operator=(rhs); // присваивает базовую часть
    // присвоение членов в производном классе, как
    // обычно,
    // отработка самоприсвоения и освобождения
    // ресурсов
    return *this;
}
```

Этот оператор начинается с явного вызова оператора присвоения базового класса, чтобы присвоить члены базовой части объекта производного. Оператор базового класса (по-видимому, правильно) отработает случай присвоения себя себе и, если нужно, освободит прежнее значение в базовой части левого операнда и присвоит новое значение правой. По завершении работы оператора продолжается выполнение всего необходимого для присвоения членов в производном классе.

Следует заметить, что конструктор или оператор присвоения производного класса может использовать соответствующую функцию базового класса независимо от того, определил ли базовый класс собственную версию этого оператора или использует синтезируемую. Например, вызов оператора Base::operator= выполняет оператор присвоения копии в классе Base. При этом несущественно, определяется ли этот оператор классом Base явно или синтезируется компилятором.

Деструктор производного класса

Помните, переменные-члены объекта неявно удаляются после завершения выполнения тела деструктора (см. раздел 13.1.3). Точно так же части базового класса объекта тоже удаляются неявно. В результате, в отличие от конструкторов и операторов присвоения, производный деструктор отвечает за освобождение только тех ресурсов, которые зарезервировал производный класс:

```
class D: public Base {  
public:  
    // Base::~Base вызывается автоматически  
    ~D() { /* освободить члены производного класса */  
}  
};
```

Объекты удаляются в порядке, противоположном их созданию: сначала выполняется деструктор производного класса, а затем деструкторы базового класса, назад по иерархии наследования.

Вызовы виртуальных функций в конструкторах и деструкторах

Как уже упоминалось, сначала создается часть базового класса в объекте производного. Пока выполняется конструктор базового класса, производная часть объекта остается неинициализированной. Точно так же производные объекты удаляются в обратном порядке, чтобы при выполнении деструктора базового класса производная часть уже была удалена. В результате на момент выполнения членов базового класса объект оказывается в незавершенном состоянии.

Чтобы приспособиться к этой незавершенности, компилятор рассматривает объект как изменяющий свой тип во время создания или удаления. Таким образом, во время создания объекта он считается объектом того же класса, что и конструктор; вызовы виртуальной функции будут связаны так, как будто у объекта тот же тип, что и у самого конструктора. Аналогично для деструктора. Эта привязка относится к виртуальным функциям, вызванным непосредственно или косвенно, из функции, которую вызывает конструктор (или деструктор).

Чтобы понять это поведение, рассмотрим, что произошло бы, если бы версия виртуальной функции производного класса была вызвана из конструктора базового класса. Эта виртуальная функция, вероятно, обратится к членам производного объекта. В конце концов, если бы

виртуальная функция не должна была использовать члены производного объекта, то производный класс, вероятно, мог бы использовать ее версию в базовом классе. Но во время выполнения конструктора базового класса эти члены остаются неинициализированными. Если бы такой доступ был разрешен, то работа программы, вероятно, закончилась бы катастрофически.



Если конструктор или деструктор вызывает виртуальную функцию, то выполняемая версия будет соответствовать типу самого конструктора или деструктора.

Упражнения раздела 15.7.3

Упражнение 15.26. Определите для классов `Quote` и `Bulk_quote` функции-члены управления копированием, осуществляющие те же действия, что и синтезируемые версии. Снабдите их и другие конструкторы операторами вывода, идентифицирующими выполняемую функцию. Напишите программу с использованием этих классов и укажите, какие объекты будут созданы и удалены. Сравните свои предположения с выводом и продолжите экспериментировать, пока ваши предположения не станут правильными.



15.7.4. Унаследованные конструкторы

По новому стандарту производный класс может многократно использовать конструкторы, определенные его прямым базовым классом. Хотя, как будет продемонстрировано далее, такие конструкторы не наследуются в обычном смысле этого слова, о них, тем не менее, говорят как об унаследованных. По тем же причинам, по которым класс может инициализировать только свой прямой базовый класс, класс может наследовать конструкторы только от своего прямого базового класса. Класс не может унаследовать стандартный конструктор, конструктор копий и перемещения. Если производный класс не определяет эти конструкторы сам, то компилятор синтезирует их, как обычно.

Производный класс наследует конструкторы своего базового класса

при помощи объявления `using`, в котором указан его (прямой) базовый класс. В качестве примера можно переопределить класс `Bulk_quote` (см. раздел 15.4) так, чтобы он унаследовал конструкторы от класса `Disc_quote`:

```
class Bulk_quote : public Disc_quote {  
public:  
    using Disc_quote::Disc_quote; // наследует  
    конструкторы Disc_quote  
    double net_price(std::size_t) const;  
};
```

Обычно объявление `using` просто делает имя видимым в текущей области видимости. Применительно к конструктору объявление `using` приводит к созданию компилятором кода. Компилятор создает в производном классе конструктор, соответствующий каждому конструктору в базовом классе. Таким образом, для каждого конструктора в базовом классе компилятор создает в производном классе конструктор с таким же списком параметров.

Эти созданные компилятором конструкторы имеют такую форму:

`производный(параметры) : базовый(аргументы) { }`

где *производный* — имя производного класса; *базовый* — имя базового класса; *параметры* — список параметров конструктора; *аргументы* передают параметры из конструктора производного класса в конструктор базового. В классе `Bulk_quote` унаследованный конструктор был бы эквивалентен следующему:

```
Bulk_quote(const std::string& book, double price,  
           std::size_t qty, double disc):  
    Disc_quote(book, price, qty, disc) {}
```

Если у производного класса есть какие-нибудь собственные переменные-члены, они инициализируются по умолчанию (см. раздел 7.1.4).

Характеристики унаследованного конструктора

В отличие от объявлений `using` для обычных членов, объявление `using` для конструктора не изменяет уровень доступа унаследованного конструктора (конструкторов). Например, независимо от того, где расположено объявление `using`, закрытый конструктор в базовом классе остается закрытым в производном; то же относится к защищенным и открытым конструкторам.

Кроме того, объявление `using` не может использовать определение как

`explicit` или `constexpr`. Если конструктор объявлен как `explicit` (см. раздел 7.5.4) или `constexpr` (см. раздел 7.5.6) в базовом классе, у унаследованного конструктора будет то же свойство.

Если у конструктора базового класса есть аргументы по умолчанию (см. раздел 6.5.1), они не наследуются. Вместо этого производный класс получает несколько унаследованных конструкторов, в которых каждый параметр с аргументом по умолчанию благополучно пропущен. Например, если у базового класса будет конструктор с двумя параметрами, у второго из которых будет аргумент по умолчанию, то производный класс получит два конструктора: один с обоими параметрами (и никакого аргумента по умолчанию) и второй конструктор с одним параметром, соответствующим левому параметру без аргумента по умолчанию в базовом классе.

Если у базового класса есть несколько конструкторов, то за двумя исключениями производный класс унаследует каждый из конструкторов своего базового класса. Первое исключение — производный класс может унаследовать некоторые конструкторы и определить собственные версии других конструкторов. Если производный класс определяет конструктор с теми же параметрами, что и конструктор в базовом классе, то этот конструктор не наследуется. Конструктор, определенный в производном классе, используется вместо унаследованного конструктора.

Второе исключение — стандартный конструктор, конструктор копий и конструктор перемещения не наследуются. Эти конструкторы синтезируются с использованием обычных правил. Унаследованный конструктор не рассматривается как пользовательский конструктор. Поэтому у класса, который содержит только унаследованные конструкторы, будет синтезируемый стандартный конструктор.

Упражнения раздела 15.7.4

Упражнение 15.27. Переопределите свой класс `Bulk_quote` так, чтобы унаследовать его конструкторы.



15.8. Контейнеры и наследование

При использовании контейнера для хранения объектов из иерархии наследования их обычно хранят косвенно. Нельзя поместить объекты связанных наследованием типов непосредственно в контейнер, поскольку нет никакого способа определить контейнер, содержащий элементы разных типов.

В качестве примера определим вектор, содержащий несколько объектов для книг, которые клиент желает купить. Вполне очевидно, что не получится использовать вектор, содержащий объекты класса `Bulk_quote`. Нельзя преобразовать объекты класса `Quote` в объекты класса `Bulk_quote` (см. раздел 15.2.3), поэтому объекты класса `Quote` в этот вектор поместить не получится.

Может быть и не так очевидно, но вектор объектов типа `Quote` также нельзя использовать. В данном случае можно поместить объекты класса `Bulk_quote` в контейнер, но эти объекты перестанут быть объектами класса `Bulk_quote`:

```
vector<Quote> basket;
basket.push_back(Quote("0-2 01-82 4 7 0-1", 50));
// ok, но в basket копируется только часть Quote
объекта
basket.push_back(Bulk_quote("0-201-54848-8",      50,
10, .25));
// вызов версии, определенной в Quote, выводит 750,
т. е. 15 * $50
cout << basket.back().net_price(15) << endl;
```

Элементами вектора `basket` являются объекты класса `Quote`. Когда в вектор добавляется объект класса `Bulk_quote`, его производная часть игнорируется (см. раздел 15.2.3).



Поскольку при присвоении объекту базового класса объект производного класса усекается, контейнеры не очень удобны для хранения объектов разных классов, связанных наследственными отношениями.

Помещайте в контейнеры указатели (интеллектуальные), а не объекты

Когда необходим контейнер, содержащий объекты, связанные наследованием, как правило, определяют контейнер указателей (предпочтительно интеллектуальных (см. раздел 12.1)) на базовый класс. Как обычно, динамический тип объекта, на который указывает этот указатель, мог бы быть типом базового класса или типом, производным от него:

```
vector<shared_ptr<Quote>> basket;
basket.push_back( make_shared<Quote>( "0-201-82470-
1", 50));
basket.push_back(
    make_shared<Bulk_quote>( "0-201-54848-8", 50, 10,
.25));
// вызов версии, определенной в Quote, выводит
562.5,
// т. е. со скидкой, меньше, чем 15 * $50
cout << basket.back() ->net_price(15) << endl;
```

Поскольку вектор `basket` содержит указатели `shared_ptr`, для получения объекта, функция `net_price()` которого выполнится, следует обратиться к значению, возвращенному функцией `basket.back()`. Для этого в вызове функции `net_price()` используется оператор `->`. Как обычно, вызываемая версия функции `net_price()` зависит от динамического типа объекта, на который указывает этот указатель.

Следует заметить, что вектор `basket` был определен как `shared_ptr<Quote>`, все же во втором вызове функции `push_back()` был передан указатель на объект класса `Bulk_quote`. Подобно тому, как можно преобразовать обычный указатель на производный тип в указатель на тип базового класса (см. раздел 15.2.2), можно также преобразовать интеллектуальный указатель на производный тип в интеллектуальный указатель на тип базового класса. Таким образом, вызов функции `make_shared<Bulk_quote>()` возвращает объект `shared_ptr<Bulk_quote>`, в который преобразуется `shared_ptr<Quote>` при вызове функции `push_back()`. В результате, несмотря на внешний вид, у всех элементов вектора `basket` будет тот же тип.

Упражнения раздела 15.8

Упражнение 15.28. Определите вектор для содержания объектов класса `Quote`, но поместите в него объекты класса `Bulk_quote`. Вычислите общую сумму результатов вызова функции `net_price()` для всех элементов вектора.

Упражнение 15.29. Повторите предыдущую программу, но на сей раз храните указатели `shared_ptr` на объекты типа `Quote`. Объясните различие в сумме данной версии программы и предыдущей. Если никакой разницы нет, объясните почему.



15.8.1. Разработка класса **Basket**

Ирония объектно-ориентированного программирования на языке C++ в том, что невозможно использовать объекты непосредственно. Вместо них приходится использовать указатели и ссылки. Поскольку указатели усложняют программы, зачастую приходится определять вспомогательные классы, чтобы избежать осложнений. Для начала определим класс, представляющий корзину покупателя:

```
class Basket {
public:
    // Basket использует синтезируемый стандартный
    // конструктор и
    // функции-члены управления копированием
    void add_item( const std::shared_ptr<Quote> &sale)
    { items.insert(sale); }
    // выводит общую стоимость каждой книги и общий
    // счет для всех
    // товаров в корзинке
    double total_receipt( std::ostream&) const;
private:
    // функция сравнения shared_ptr, необходимая
    // элементам
    // набора multiset
    static bool compare( const std::shared_ptr<Quote>
    &lhs,
                           const std::shared_ptr<Quote>
    &rhs)
    { return lhs->isbn() < rhs->isbn(); }
    // набор multiset содержит несколько стратегий
    // расценок,
    // упорядоченных по сравниваемому элементу
    std::multiset<std::shared_ptr<Quote>,
    decltype(compare) *>
    items{compare};
}
```

Для хранения транзакций класс использует контейнер `multiset` (см.

раздел 11.2.1), позволяющий содержать несколько транзакций по той же книге, чтобы все транзакции для данной книги находились вместе (см. раздел 11.2.2).

Элементами контейнера `multiset` будут указатели `shared_ptr`, и для них нет оператора "меньше". В результате придется предоставить собственный оператор сравнения для упорядочивания элементов (см. раздел 11.2.2). Здесь определяется закрытая статическая функция-член `compare()`, сравнивающая `isbn` объектов, на которые указывают указатели `shared_ptr`. Инициализируем контейнер `multiset` с использованием этой функции сравнения и внутриклассового инициализатора (см. раздел 7.3.1):

```
// набор multiset содержит несколько стратегий
расценок,
// упорядоченных по сравниваемому элементу
std::multiset<std::shared_ptr<Quote>,
decltype(compare) *>
items{ compare};
```

Это объявление может быть трудно понять, но, читая его слева направо, можно заметить, что определяется контейнер `multiset` указателей `shared_ptr` на объекты класса `Quote`. Для упорядочивания элементов контейнер `multiset` будет использовать функцию с тем же типом, что и функция-член `compare()`. Элементами контейнера `multiset` будут объекты `items`, которые инициализируются для использования функции `compare()`.

Определение членов класса Basket

Класс `Basket` определяет только две функции. Функция-член `add_item()` определена в классе. Она получает указатель `shared_ptr` на динамически созданный объект класса `Quote` и помещает его в контейнер `multiset`. Вторая функция-член, `total_receipt()`, выводит полученный счет для содержимого корзины и возвращает цену за все элементы в ней:

```
double Basket::total_receipt(ostream &os) const {
    double sum = 0.0; // содержит текущую сумму

    // iter ссылается на первый элемент в пакете
    // элементов с тем же ISBN
    // upper_bound() возвращает итератор на элемент
```

сразу после

```
// конца этого пакета
for (auto iter = items.cbegin();
     iter != items.cend();
     iter = items.upper_bound( *iter) ) {
    // известно, что в Basket есть по крайней мере
один элемент
    // с этим ключом
    // вывести строку для элемента этой книги
    sum += print_total(os, **iter,
items.count( *iter));
}
os << "Total Sale: " << sum << endl; // вывести в
конце общий счет
return sum;
}
```

Цикл `for` начинается с определения и инициализации итератора `iter` на первый элемент контейнера `multiset`. Условие проверяет, не равен ли `iter` значению `items.cend()`. Если да, то обработаны все покупки и цикл `for` завершается. В противном случае обрабатывается следующая книга.

Интересный момент — выражение "инкремента" в цикле `for`. Это не обычный цикл, читающий каждый элемент и перемещающий итератор `iter` на следующий. При вызове функции `upper_bound()` (см. раздел 11.3.5) он перескакивает через все элементы, которые соответствуют текущему ключу. Вызов функции `upper_bound()` возвращает итератор на элемент сразу после последнего с тем же ключом, что и `iter`. Возвращаемый итератор обозначает или конец набора, или следующую книгу.

Для вывода подробностей по каждой книге в корзине в цикле `for` происходит вызов функции `print_total()` (см. раздел 15.1):

```
sum += print_total(os, **iter, items.count( *iter));
```

Аргументами функции `print_total()` являются поток `ostream` для записи, обрабатываемый объект `Quote` и счет. При обращении к значению итератора `iter` возвращается указатель `shared_ptr`, указывающий на объект, который предстоит вывести. Чтобы получить этот объект, следует обратиться к значению этого указателя `shared_ptr`. Таким образом, выражение `**iter` возвращает объект класса `Quote` (или класса

производного от него). Для выяснения количества элементов в контейнере `multiset` с тем же ключом (т.е. с тем же ISBN) используется его функция-член `count()` (см. раздел 11.3.5).

Как уже упоминалось, функция `print_total()` осуществляет вызов виртуальной функции `net_price()`, поэтому полученная цена зависит от динамического типа `**iter`. Функция `print_total()` выводит общую сумму для данной книги и возвращает вычисленную общую стоимость. Результат добавляется в переменную `sum`, которая выводится после завершения цикла `for`.

Сокрытие указателей

Пользователи класса `Basket` все еще должны иметь дело с динамической памятью, поскольку функция `add_item()` получает указатель `shared_ptr`. В результате пользователи вынуждены писать код так:

```
Basket bsk;
bsk.add_item( make_shared<Quote>("123", 45));
bsk.add_item( make_shared<Bulk_quote>("345", 45, 3,
.15));
```

На следующем этапе переопределим функцию `add_item()` так, чтобы она получала объект класса `Quote` вместо указателя `shared_ptr`. Эта новая версия функции `add_item()` отработает резервирование памяти так, чтобы пользователи больше не должны были делать это сами. Определим две ее версии: одна будет копировать переданный ей объект, а другая перемещать его (см. раздел 13.6.3):

```
void add_item( const Quote& sale); // копирует
переданный объект
void add_item( Quote&& sale); // перемещает
переданный объект
```

Единственная проблема в том, что функция `add_item()` не знает, какой тип резервировать. При резервировании памяти функция `add_item()` скопирует (или переместит) свой параметр `sale`. Выражение `new` будет выглядеть примерно так:

```
new Quote( sale)
```

К сожалению, это выражение будет неправильным: оператор `new` резервирует объект запрошенного типа. Оно резервирует объект типа `Quote` и копирует часть `Quote` параметра `sale`. Но если переданный параметру `sale` объект будет иметь тип `Bulk_quote`, то он будет усечен.



Имитация виртуального копирования

Эту проблему можно решить, снабдив класс `Quote` виртуальной функцией-членом, резервирующей его копию.

```
class Quote {
public:
    // виртуальная функция, возвращающая динамически
    // созданную копию
    // эти члены используют квалификаторы ссылки;
раздел 13.6.3
    virtual Quote* clone() const & { return new
Quote(*this); }
    virtual Quote* clone() &&
    { return new Quote(std::move(*this)); }
    // другие члены как прежде
};

class Bulk_quote : public Quote {
    Bulk_quote* clone() const & { return new
Bulk_quote(*this); }
    Bulk_quote* clone() &&
    { return new Bulk_quote(std::move(*this)); }
    // другие члены, как прежде
};
```

Поскольку функция `add_item()` имеет версии копирования и перемещения, были определены версии l- и r-значения функции `clone()` (см. раздел 13.6.3). Каждая функция `clone()` резервирует новый объект ее собственного типа. Функция-член константной ссылки на l-значение копирует себя во вновь зарезервированный объект; функция-член ссылки на r-значение перемещает свои данные.

Используя функцию `clone()`, довольно просто написать новые версии функции `add_item()`:

```
class Basket {
public:
    void add_item(const Quote& sale) // копирует
переданный объект
    {
        items.insert(std::shared_ptr<Quote>
```

```

(sale.clone( )) ; }

void add_item( Quote&& sale) // перемещает
переданный объект
{ items.insert(
    std::shared_ptr<Quote>
( std::move( sale).clone( )) );
// другие члены, как прежде
}

```

Как и сама функция `add_item()`, функция `clone()` перегружается на основании того, вызвана ли она для l- или r-значения. Таким образом, первая версия функции `add_item()` вызывает константную версию l-значения функции `clone()`, а вторая версия вызывает версию ссылки на r-значение. Обратите внимание, что хотя в версии r-значения типом параметра `sale` является ссылка на r-значение, сам параметр `sale` (как и любая другая переменная) является l-значением (см. раздел 13.6.1). Поэтому для привязки ссылки на r-значение к параметру `sale` вызывается функция `move()`.

Наша функция `clone()` является также виртуальной. Будет ли выполнена функция из класса `Quote` или `Bulk_quote`, зависит (как обычно) от динамического типа параметра `sale`. Независимо от того, копируются или перемещаются данные, функция `clone()` возвращает указатель на вновь зарезервированный объект его собственного типа. С этим объектом связывается указатель `shared_ptr`, и вызывается функция `insert()` для добавления этого вновь зарезервированного объекта к `items`. Обратите внимание: так как указатель `shared_ptr` поддерживает преобразование производного класса в базовый (см. раздел 15.2.2), указатель `shared_ptr<Quote>` можно привязать к `Bulk_quote*`.

Упражнения раздела 15.8.1

Упражнение 15.30. Напишите собственную версию класса `Basket` и используйте ее для вычисления цены за те же транзакции, что и в предыдущих упражнениях.

15.9. Возвращаясь к запросам текста

В качестве последнего примера наследования дополним приложение текстового запроса из раздела 12.3. Написанные в этом разделе классы позволяют искать вхождения данного слова в файле. Дополним эту систему возможностью создавать более сложные запросы. В этих примерах запросы будут выполняться к тексту следующей истории:

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?"

Система должна поддерживать следующие запросы.

- Запросы слов находят все строки, соответствующие заданной строке:

Executing Query for: Daddy

Daddy occurs 3 times

(line 2) Her Daddy says when the wind blows

(line 7) "Daddy, shush, there is no such thing,"

(line 10) Shyly, she asks, "I mean, Daddy, is
there?"

- Инверсный запрос с использованием оператора ~ возвращает строки, которые не содержат заданную строку:

Executing Query for: ~(Alice)

~(Alice) occurs 9 times

(line 2) Her Daddy says when the wind blows

(line 3) through her hair, it looks almost alive,

(line 4) like a fiery bird in flight.

...

- Запросы ИЛИ с использованием оператора | возвращают строки, содержащие любую из двух заданных строк:

Executing Query for: (hair | Alice)

(hair | Alice) occurs 2 times

(line 1) Alice Emma has long flowing red hair,

(line 3) through her hair, it looks almost alive,

- Запросы И с использованием оператора & возвращают строки, содержащие обе заданные строки:

Executing query for: (hair & Alice)

(hair & Alice) occurs 1 time

(line 1) Alice Emma has long flowing red hair.

Кроме того, нужна возможность объединить эти операторы так fiery & bird | wind

Для обработки составных выражений, таких как в этом примере, будут использованы обычные правила приоритета C++ (см. раздел 4.1.2). Таким образом, этому запросу соответствует строка, в которой присутствуют слова fiery и bird или слово wind:

Executing Query for: ((fiery & bird) | wind)

((fiery & bird) | wind) occurs 3 times

(line 2) Her Daddy says when the wind blows

(line 4) like a fiery bird in flight.

(line 5) A beautiful fiery bird, he tells her,

В отображаемом результате для указания способа интерпретации запроса используются круглые скобки. Подобно первоначальной реализации, система не должна отображать одинаковые строки несколько раз.

15.9.1. Объектно-ориентированное решение

Для представления запросов на поиск слов вполне логично было бы использовать класс TextQuery (см. раздел 12.3.2), а другие классы запросов можно было бы получить как производные от этого класса.

Однако такой подход неверен. Концептуально инверсный запрос не является разновидностью запроса на поиск слова. Инверсный запрос — это скорее запрос типа "имеет" (запрос на поиск слова или любой другой тип запроса), результат которого интерпретируется негативно.

Исходя из этого можно сделать вывод, что разные виды запросов следует оформить как независимые классы, которые совместно используют общий базовый класс:

WordQuery // Daddy

NotQuery // ~Alice

OrQuery // hair | Alice

AndQuery // hair & Alice

Эти классы будут иметь только две функции.

- Функция `eval()`, получающая объект класса `TextQuery` и возвращающая объект класса `QueryResult`. Для поиска запрошенной строки функция `eval()` будет использовать переданный объект класса `TextQuery`.
- Функция `rep()`, возвращающая строковое представление базового запроса. Эту функцию использует функция `eval()` для создания объекта класса `QueryResult`, представляющего соответствие, а также оператор вывода, отображающий выражение запроса.

Абстрактный базовый класс

Как уже упоминалось, все четыре типа запроса не связаны друг с другом наследованием; концептуально они элементы одного уровня. Каждый класс использует тот же интерфейс, а значит, для представления этого интерфейса следует определить абстрактный базовый класс (см. раздел 15.4). Назовем этот абстрактный базовый класс `Query_base`, поскольку он должен служить корневым классом иерархии запроса.

Ключевая концепция. Наследование или композиция

Проектирование иерархии наследования — это достаточно сложная тема, которая выходит за рамки данного вводного курса. Однако имеет смысл упомянуть об одном достаточно важном факторе проектирования, с которым должен быть знаком каждый программист.

При определении класса как открыто производного от другого производный и базовый классы реализуют взаимоотношения типа "является" (*is a*). В хорошо проработанных иерархиях объекты открыто унаследованных классов применимы везде, где ожидается объект базового класса.

Еще одним популярным способом взаимоотношений классов является принцип "имеет" (*has a*). Типы, связанные отношениями "имеет", подразумевают принадлежность.

В рассматриваемом примере с книжным магазином базовый класс представляет концепцию книги, продаваемой по предусмотренной цене, а класс `Bulk_quote` "является" конкретной книгой, продаваемой по розничной цене с определенной стратегией скидок. Классы приложения книжного магазина "имеют" цену и ISBN.

Класс `Query_base` определит функции `eval()` и `rep()` как чистые виртуальные (см. раздел 15.4). Каждый из классов, представляющих специфический вид запроса, должен переопределить эти функции. Классы

WordQuery и NotQuery унаследуем непосредственно от класса Query_base. У классов AndQuery и OrQuery будет одна общая особенность, которой не будет у остальных классов в системе: у каждого будет по два операнда. Для моделирования этой особенности определим другой абстрактный базовый класс, BinaryQuery, представляющий запросы с двумя operandами. Классы AndQuery и OrQuery наследуются от класса BinaryQuery, который в свою очередь наследуется от класса Query_base. Результатом этих решений будет проект классов, представленный на рис. 15.2.

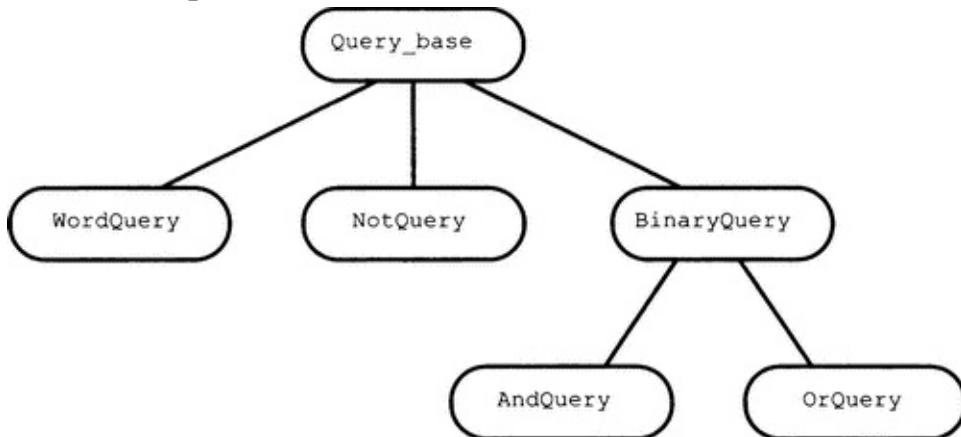


Рис. 15.2. Иерархия наследования Query_base

Сокрытие иерархии в классе интерфейса

Рассматриваемая программа будет отрабатывать запросы, а не создавать их. Но чтобы запустить программу на выполнение, необходимо определить способ создания запроса. Проще всего сделать это непосредственно в коде при помощи выражения C++. Например, чтобы создать описанный ранее составной запрос, можно использовать следующий код:

```
Query q = Query("fiery") & Query("bird") | Query("wind");
```

Это довольно сложное описание неявно предполагает, что код пользовательского уровня не будет использовать унаследованные классы непосредственно. Вместо этого будет создан класс интерфейса по имени Query (Запрос), который и скроет иерархию. Класс Query будет хранить указатель на класс Query_base. Этот указатель будет связан с объектом типа, производного от класса Query_base. Класс Query будет предоставлять те же функции, что и классы Query_base: функцию eval() для обработки соответствующего запроса и функцию rep() для

создания строковой версии запроса. В нем также будет определен перегруженный оператор вывода, чтобы отображать соответствующий запрос.

Пользователи будут создавать объекты класса `Query_base` и работать с ними только косвенно, через функции объектов класса `Query`. Для класса `Query`, наряду с получающим строку конструктором, определим три перегруженных оператора. Каждая из этих функций будет динамически резервировать новый объект типа, производного от класса `Query_base`:

- Оператор `&` создает объект класса `Query`, связанный с новым объектом класса `AndQuery`.
- Оператор `|` создает объект класса `Query`, связанный с новым объектом класса `OrQuery`.
- Оператор `~` создает объект класса `Query`, связанный с новым объектом класса `NotQuery`.
- Конструктор класса `Query`, получающий строку и создающий новый объект класса `WordQuery`.

Как работают эти классы

Следует понять, что работа этого приложения состоит в основном из построения объектов для представления запросов пользователя. Например, приведенное выше выражение создает коллекцию взаимодействовавших объектов, представленных на рис. 15.3.

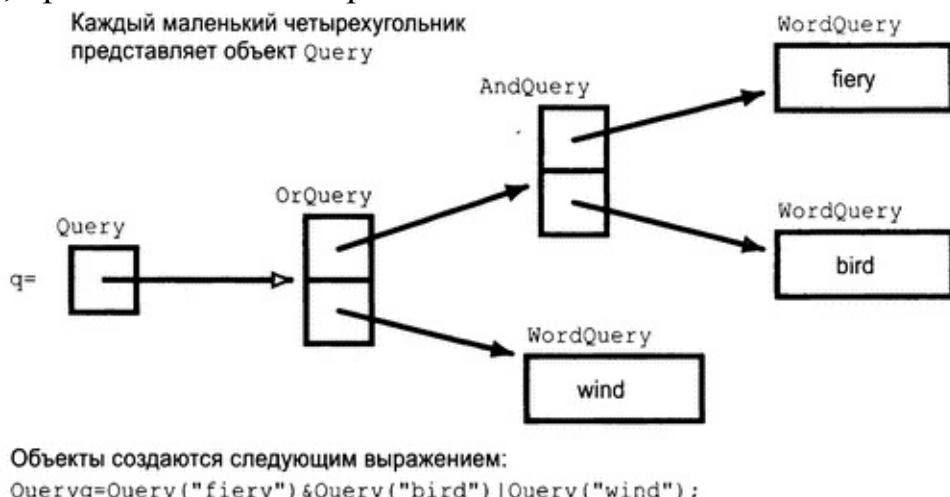


Рис. 15.3. Объекты, созданные выражениями запросов

Как только создано дерево объектов, обработка (или отображение) данного запроса сводится к простому процессу (осуществляемому

компилятором), который, следуя по линиям, опрашивает каждый объект дерева, чтобы выполнить (или отобразить) необходимые действия. Например, если происходит вызов функции `eval()` объекта `q` (т.е. корневого класса дерева), функция `eval()` опросит объект класса `OrQuery`, на который он указывает. Обработка этого объекта класса `OrQuery` приведет к вызову функции `eval()` для двух его операндов, что, в свою очередь, приведет к вызову функции `eval()` для объектов классов `AndQuery` и `WordQuery`, которые осуществляют поиск слова `wind`. Обработка объекта класса `AndQuery`, в свою очередь, приведет к обработке двух его объектов класса `WordQuery`, создав результаты для слов `fiery` и `bird` соответственно.

Новичкам в объектно-ориентированном программировании зачастую трудней всего разобраться в проекте программы. Но как только проект станет абсолютно понятен, его реализация не вызывает проблем. Чтобы проще было понять суть проекта, все используемые в этом примере классы были обобщены в табл. 15.1.

Таблица 15.1. Обзор проекта программы

Классы и операторы интерфейса программы запросов	
<code>TextQuery</code>	Класс, который читает указанный файл и создает карту поиска. Этот класс предоставляет функцию поиска <code>query()</code> , которая получает строковый аргумент и возвращает объект класса <code>QueryResult</code> , представляющий строки, в которых присутствует ее аргумент (см. раздел 12.3.2)
<code>QueryResult</code>	Класс, содержащий результаты вызова функции <code>query()</code> (см. раздел 12.3.2)
<code>Query</code>	Класс интерфейса, указывающий на объект типа, производного от класса <code>Query_base</code>
<code>Query q(s)</code>	Связывает объект <code>q</code> класса <code>Query</code> с новым объектом класса <code>WordQuery</code> , содержащим строку <code>s</code>
<code>q1 & q2</code>	Возвращает объект класса <code>Query</code> , связанный с новым объектом класса <code>AndQuery</code> , который содержит объекты <code>q1</code> и <code>q2</code>
<code>q1 q2</code>	Возвращает объект класса <code>Query</code> , связанный с новым объектом класса <code>OrQuery</code> , содержащим объекты <code>q1</code> и <code>q2</code>
<code>~q</code>	Возвращает объект класса <code>Query</code> , связанный с новым объектом класса <code>NotQuery</code> , содержащим объект <code>q</code>
Классы реализации программы запросов	
<code>Query_base</code>	Абстрактный класс, базовый для классов запроса

WordQuery	Класс, производный от класса <code>Query_base</code> , который ищет указанное слово
NotQuery	Класс, производный от класса <code>Query_base</code> , представляющий набор строк, в которых указанный операнд <code>Query</code> отсутствует
BinaryQuery	Абстрактный базовый класс, производный от класса <code>Query_base</code> , который представляет запросы с двумя operandами типа <code>Query</code>
OrQuery	Класс, производный от класса <code>BinaryQuery</code> , который возвращает набор номеров строк, в которых присутствует хотя бы один из operandов
AndQuery	Класс, производный от класса <code>BinaryQuery</code> , который возвращает набор номеров строк, в которых присутствуют оба операнда

Упражнения раздела 15.9.1

Упражнение 15.31. При условии, что `s1`, `s2`, `s3` и `s4` являются строками укажите, какие объекты создаются в следующих выражениях:

- (a) `Query(s1) | Query(s2) & ~ Query(s3);`
- (b) `Query(s1) | (Query(s2) & ~ Query(s3));`
- (c) `(Query(s1) & (Query(s2)) | (Query(s3) & Query(s4)) ;`

15.9.2. Классы `Query_base` и `Query`

Начнем реализацию с определения класса `Query_base`:

```
// абстрактный класс, являющийся базовым для
конкретных типов запроса;
// все члены закрыты
class Query_base {
    friend class Query;
protected:
    using line_no = TextQuery::line_no; // 
используется в функциях eval()
    virtual ~Query_base() = default;
private:
    // eval() возвращает соответствующий запросу
QueryResult
    virtual QueryResult eval(const TextQuery&) const =
0;
    // rep() строковое представление запроса
    virtual std::string rep() const = 0;
};
```

Обе функции, `eval()` и `rep()`, являются чистыми виртуальными, что делает класс `Query_base` абстрактным базовым (см. раздел 15.4). Поскольку класс `Query_base` не предназначен для пользователей и непосредственного использования в производных классах, у него нет открытых членов. Класс `Query_base` будет использоваться только через объекты класса `Query`. Класс предоставляет дружественные отношения классу `Query`, поскольку его члены вызывают виртуальные функции класса `Query_base`.

Защищенный член `line_no` будет использоваться в функциях `eval()`. Деструктор также будет защищен, поскольку он используется (неявно) деструкторами в производных классах.

Класс Query

Класс `Query` предоставляет интерфейс к иерархии наследования `Query_base` и скрывает ее. Каждый объект класса `Query` содержит указатель `shared_ptr` на соответствующий объект класса `Query_base`. Поскольку класс `Query` — единственный интерфейс к классам иерархии `Query_base`, он должен определить собственные версии функций `eval()` и `rep()`.

Конструктор `Query()`, получающий строку, создаст новый объект класса `WordQuery` и свяжет его указатель-член `shared_ptr` с этим недавно созданным объектом. Операторы `&`, `|` и `~` создают объекты `AndQuery`, `OrQuery` и `NotQuery` соответственно. Эти операторы возвращают объект класса `Query`, связанный с созданным им объектом. Для поддержки этих операторов класс `Query` нуждается в конструкторе, получающем указатель `shared_ptr` на класс `Query_base` и сохраняющем его. Сделаем этот конструктор закрытым, поскольку объекты класса `Query_base` не предназначены для определения общим пользовательским кодом. Так как этот конструктор является закрытым, операторы следует сделать дружественными.

Исходя из приведенного выше проекта, сам класс `Query` довольно прост:

```
// класс интерфейса для взаимодействия с иерархией
// наследования Query_base
class Query {
    // эти операторы должны обращаться к указателю
    shared_ptr
    friend Query operator~(const Query &);
```

```

        friend Query operator| ( const Query&, const
Query& );
        friend Query operator&( const Query&, const
Query& );
public:
    Query( const std::string& ); // создает новый
WordQuery
    // функции интерфейса: вызывают соответствующий
оператор Query_base
    QueryResult eval( const TextQuery &t) const
    { return q->eval( t); }
    std::string rep() const { return q->rep(); }
private:
    Query( std::shared_ptr<Query_base> query): q( query)
{ }
    std::shared_ptr<Query_base> q;
} ;

```

Начнем с объявления дружественных операторов, создающих объекты класса `Query`. Эти операторы должны быть друзьями, чтобы использовать закрытый конструктор.

В открытом интерфейсе для класса `Query` объявляется, но еще не может быть определен получающий строку конструктор. Этот конструктор создает объект класса `WordQuery`, поэтому невозможно определить этот конструктор, пока не определен сам класс `WordQuery`.

Два других открытых члена представляют интерфейс для класса `Query_base`. В каждом случае оператор класса `Query` использует свой указатель класса `Query_base` для вызова соответствующей (виртуальный) функции класса `Query_base`. Фактически вызываемая версия определяется во время выполнения и будет зависеть от типа объекта, на который указывает указатель `q`.



Оператор вывода класса `Query`

Оператор вывода — хороший пример того, как работает вся система запросов:

```

std::ostream &
operator<<( std::ostream &os, const Query &query) {

```

```

    // Query::rep() осуществляет виртуальный вызов
    // через свой
    // указатель Query_base на rep()
    return os << query.rep();
}

```

При выводе объекта класса `Query` оператор вывода вызывает (открытую) функцию-член `rep()` класса `Query`. Эта функция осуществляет виртуальный вызов через свой указатель-член функции-члена `rep()` объекта, на который указывает данный объект класса `Query`.

```

Query andq = Query(sought1) & Query(sought2);
cout << andq << endl;

```

Таким образом, когда в коде встречается оператор вывода, он вызывает функцию `Query::rep()` объекта `andq`. Функция `Query::rep()` в свою очередь осуществляет виртуальный вызов через свой указатель класса `Query_base` на версию функции `rep()` класса `Query_base`. Поскольку объект `andq` указывает на объект класса `AndQuery`, этот вызов выполнит функцию `AndQuery::rep()`.

Упражнения раздела 15.9.2

Упражнение 15.32. Что будет при копировании, перемещении, присвоении и удалении объекта класса `Query`?

Упражнение 15.33. А объектов класса `Query_base`?

15.9.3. Производные классы

Самая интересная часть классов, производных от класса `Query_base`, в том, как они представляются. Класс `WordQuery` проще всех. Его задача — хранение искомого слова.

Другие классы работают на одном или двух операндах. У класса `NotQuery` один operand, а у классов `AndQuery` и `OrQuery` — по два. Операндами в каждом из этих классов могут быть объекты любого из реальных классов, производных от класса `Query_base`: `NotQuery` может быть применен к `WordQuery`, как и `AndQuery`, `OrQuery` или `NotQuery`. Для обеспечения такой гибкости операнды следует хранить как указатели на класс `Query_base`. Таким образом, можно привязать указатель на любой необходимый реальный класс.

Но вместо того, чтобы хранить указатель на класс `Query_base`, классы будут сами использовать объект `Query`. Подобно тому, как

пользовательский код упрощается при использовании класса интерфейса, можно упростить код собственного класса, используя тот же класс.

Теперь, когда конструкция этих классов известна, их можно реализовать.

Класс WordQuery

Класс WordQuery отвечает за поиск заданной строки. Это единственная операция, которая фактически выполняет запрос для данного объекта класса TextQuery:

```
class WordQuery: public Query_base {
    friend class Query; // Query использует
конструктор WordQuery
    WordQuery( const std::string &s) : query_word ( s) {
}
    // конкретный класс: WordQuery определяет все
унаследованные чистые
    // виртуальные функции
    QueryResult eval( const TextQuery &t) const
    { return t.query(query_word); }
    std::string rep( ) const { return query_word; }
    std::string query_word; // искомое слово
};

Подобно классу Query_base, у класса WordQuery нет открытых
```

членов; он должен сделать класс Query дружественным, чтобы позволить ему получать доступ к конструктору WordQuery().

Каждый из конкретных классов запроса должен определить унаследованные чистые виртуальные функции eval() и rep(). Обе функции определены в теле класса WordQuery: функция eval() вызывает функцию-член query() своего параметра типа TextQuery, который фактически осуществляет поиск в файле; функция rep() возвращает строку, которую данный объект класса WordQuery представляет (т.е. query_word).

Определив класс WordQuery, можно определить конструктор Query(), получающий строку:

```
inline
Query::Query( const std::string &s): q( new
WordQuery(s) ) { }
```

Этот конструктор резервирует объект класса WordQuery и

инициализирует его указатель-член так, чтобы он указывал на этот недавно созданный объект.

Класс NotQuery и оператор ~

Оператор `~` подразумевает создание объекта класса `NotQuery`, содержащего инверсный запрос:

```
class NotQuery: public Query_base {
    friend Query operator~(const Query &);

    NotQuery( const Query &q): query(q) { }

    // конкретный класс: NotQuery определяет все
    // унаследованные

    // чистые виртуальные функции
    std::string rep() const { return + query.rep() +
")"; }

    QueryResult eval( const TextQuery&) const;
    Query query;
};

inline Query operator~( const Query &operand) {
    return std::shared_ptr<Query_base>( new
NotQuery( operand));
}
```

Поскольку все члены класса `NotQuery` являются закрытыми, объявляем оператор `~` дружественным. Чтобы отобразить объект класса `NotQuery`, следует вывести символ `"~"` сопровождаемый основным запросом. Чтобы сделать приоритет очевидным для читателя, заключим запрос в скобки.

Следует заметить то, что вызов функции `rep()` объекта класса `NotQuery` в конечном счете приводит к виртуальному вызову функции собственной функции-члена `rep():query.rep()` — это невиртуальный вызов функции-члена `rep()` класса `Query`. Функция `Query::rep()` в свою очередь осуществляет вызов `q->rep()`, являющийся виртуальным вызовом через указатель `Query_base`.

Оператор `~` динамически резервирует новый объект класса `NotQuery`. Оператор `return` (неявно) использует конструктор `Query()`, получающий указатель `shared_ptr<Query_base>`. Таким образом, оператор `return` эквивалентен следующему:

```
// резервировать новый объект NotQuery
// связать новый объект NotQuery с указателем
```

```

shared_ptr<Query_base>
    shared_ptr<Query_base> tmp( new NotQuery( expr ) );
    return Query( tmp ); // использовать конструктор
Query( ), получающий
                                // указатель shared_ptr

```

Функция-член eval() достаточно сложна, поэтому реализуем ее вне тела класса. Более подробно функция eval() рассматривается в разделе 15.9.4.

Класс BinaryQuery

Класс BinaryQuery — это абстрактный базовый класс, содержащий данные, необходимые двум классам запроса, AndQuery и OrQuery, которые используют по два операнда:

```

class BinaryQuery: public Query_base {
protected:
    BinaryQuery( const Query &l, const Query &r,
std::string s):
        lhs(l), rhs(r), opSym(s) { }
    // абстрактный класс: BinaryQuery функцию eval()
не определяет
    std::string rep() const { return "(" + lhs.rep() +
" "
                                + opSym + " "
                                + rhs.rep() +
")"; }
    Query lhs, rhs; // правый и левый операнды
    std::string opSym; // имя оператора
};


```

Данными класса BinaryQuery являются два операнда запроса и символ оператора. Конструктор получает эти два операнда и символ оператора, каждый из которых он хранит в соответствующих переменных-членах.

Чтобы отобразить объект класса BinaryOperator, следует вывести выражение в скобках, состоящее из левого операнда, оператора и правого операнда. Как и в случае класса NotQuery, вызов функции rep() в конечном счете осуществляет вызов виртуальных функций rep() объектов класса Query_base, на которые указывают параметры lhs и rhs.



Класс `BinaryQuery` не переопределяет функцию `eval()`, а следовательно, наследует ее чистой виртуальной. Таким образом, класс `BinaryQuery` остается абстрактным и его объекты создавать нельзя.

Классы `AndQuery`, `OrQuery` и их операторы

Классы `AndQuery` и `OrQuery`, а также соответствующие им операторы очень похожи:

```
class AndQuery: public BinaryQuery {
    friend Query operators( const Query&, const Query& );
    AndQuery( const Query &left, const Query &right):
        BinaryQuery(left, right, "&") { }
    // конкретный класс: AndQuery наследует функцию rep(),
    // а остальные чистые виртуальные функции переопределяет
    QueryResult eval( const TextQuery&) const;
};

inline Query operator&( const Query &lhs, const Query &rhs) {
    return std::shared_ptr<Query_base>( new
AndQuery( lhs, rhs));
}

class OrQuery: public BinaryQuery {
    friend Query operator|( const Query&, const Query& );
    OrQuery( const Query &left, const Query &right):
        BinaryQuery(left, right, "|") { }
    QueryResult eval( const TextQuery&) const;
};
inline Query operator|( const Query &lhs, const Query &rhs) {
    return std::shared_ptr<Query_base>( new
```

```
OrQuery( lhs, rhs ) ;  
}
```

Эти классы объявляют соответствующий оператор дружественным и определяют конструктор, создающий их базовую часть класса `BinaryQuery` с соответствующим оператором. Они наследуют определение функции `rep()` от класса `BinaryQuery`, но каждый из них определяет собственную версию функции `eval()`.

Как и операторы `&` и `|` возвращают указатель `shared_ptr` на вновь созданный объект соответствующего типа. Этот указатель `shared_ptr` приводится к типу `Query` в операторе `return` каждого из этих операторов.

Упражнения раздела 15.9.3

Упражнение 15.34. Исходя из выражения, представленного на рис. 15.3:

- Перечислите конструкторы, задействованные при обработке этого выражения;
- Перечислите обращения к функции `rep()` из выражения `cout << q;`
- Перечислите обращения к функции `eval()` из выражения `q.eval()`.

Упражнение 15.35. Реализуйте классы `Query` и `Query_base`, включая определение функции `rep()`, но исключая определение функции `eval()`.

Упражнение 15.36. Добавьте операторы вывода в конструкторы и функции-члены `rep()`. Запустите код на выполнение, чтобы проверить свои ответы на вопросы (a) и (b) первого упражнения.

Упражнение 15.37. Какие изменения следовало бы внести в классы, если бы у производных классов были члены типа `shared_ptr<Query_base>`, а не типа `Query`?

Упражнение 15.38. Допустимы ли следующие объявления? Если нет, то почему? Если да, то что они означают?

```
BinaryQuery a = Query( "fiery" ) & Query( "bird" );  
AndQuery b = Query( "fiery" ) & Query( "bird" );  
OrQuery c = Query( "fiery" ) & Query( "bird" );
```

15.9.4. Виртуальные функции `eval()`

Функции `eval()` — основа системы запросов. Каждая из них

вызывает функцию `eval()` своего операнда (операндов), а затем применяет собственную логику вычислений: функция `eval()` класса `OrQuery` возвращает объединение результатов своих operandов, а функция `eval()` класса `AndQuery` возвращает их пересечение. Функция `eval()` класса `NotQuery` немного сложней: она должна возвращать номера строк, не входящих в набор операнда.

Для обеспечения обработки в функциях `eval()` необходимо использовать ту версию класса `QueryResult`, в который определены члены, добавленные в упражнениях раздела 12.3.2. Подразумевается, что у класса `QueryResult` есть функции-члены `begin()` и `end()`, позволяющие перебрать набор номеров строк, которые содержит объект класса `QueryResult`. Подразумевается также, что у класса `QueryResult` есть функция-член `get_file()`, возвращающая указатель `shared_ptr` на файл, к которому осуществляется запрос.



Класс `Query` использует функции-члены `begin()` и `end()`, определенные для класса `QueryResult` в упражнении 12.3.2.

Функция `OrQuery::eval()`

Функция `eval()` класса `OrQuery` объединяет наборы номеров строк, возвращенных его operandами, т.е. ее результатом является объединение результатов двух operandов.

Объект класса `OrQuery` представляет объединение результатов двух своих operandов, полученных при вызове функции-члена `eval()` каждого из них. Поскольку эти operandы являются объектами класса `Query`, вызов функции `eval()` является вызовом `Query::eval()`, который в свою очередь осуществляет виртуальный вызов функции `eval()` объекта базового класса `Query_base`. Каждый из этих вызовов возвращает объект класса `QueryResult`, представляющий номера строк, в которых присутствует его operand. Эти номера строк объединяются в новый набор:

```
// возвращает объединение наборов результатов своих operandов
```

```
QueryResult  
OrQuery::eval(const TextQuery& text) const {  
    // виртуальные вызовы через члены Query, lhs и rhs
```

```

    // вызовы eval() возвращают QueryResult для
    // каждого операнда
    auto right = rhs.eval(text), left =
lhs.eval(text);
    // копировать номера строк левого операнда в
результатирующий набор
    auto ret_lines =
        make_shared<set<line_no>>(left.begin(),
left.end());
    // вставить строки из правого операнда
    ret_lines->insert(right.begin(), right.end());
    // возвратить новый QueryResult, представляющий
объединение lhs и rhs
    return QueryResult(rep(),
ret_lines,
left.get_file());
}

```

Набор `ret_lines` инициализируется с использования того конструктора, который получает пару итераторов. Функции-члены `begin()` и `end()` класса `QueryResult` возвращают итераторы на номера строк набора. Таким образом, набор `ret_lines` создается при копировании элементов из набора `left`. Затем для вставки элементов из набора `right` вызывается функция `insert()`. После этого вызова набор `ret_lines` содержит номера строк из наборов, которые присутствуют в наборах `left` или `right`.

Функция `eval()` завершает работу, создавая и возвращая объект класса `QueryResult`, представляющий объединение соответствий. Конструктор `QueryResult()` (см. раздел 12.3.2) получает три аргумента: строку, представляющую запрос, указатель `shared_ptr` на набор соответствующих номеров строк и указатель `shared_ptr` на вектор, представляющий входной файл. Вызов функции `rep()` позволяет создать строку, а вызов функции `get_file()` — получить указатель `shared_ptr` на файл. Поскольку оба набора, `left` и `right`, относятся к тому же файлу, не имеет значения, который из них использовать для функции `get_file()`.

Функция *AndQuery* : `eval()`

Версия функции `eval()` класса `AndQuery` подобна версии класса `OrQuery`, за исключением того, что она использует библиотечный

алгоритм для поиска строк, общих для обоих запросов:

```
// возвращает пересечение наборов результатов своих
операндов
QueryResult
AndQuery::eval( const TextQuery& text) const {
    // виртуальный вызов через операнды класса Query
для получения
    // результирующих наборов для operandov
    auto left      = lhs.eval(text), right      =
rhs.eval(text);
    // набор для хранения пересечения left и right
    auto ret_lines = make_shared<set<line_no>>();
    // выводит пересечение двух диапазонов в итератор
назначения
    // итератор назначения в этом вызове добавляет
элементы в ret
    set_intersection(left.begin(), left.end(),
                    right.begin(), right.end(),
                    inserter(*ret_lines, ret_lines-
>begin()));
    return QueryResult(rep(), ret_lines,
left.get_file());
}
```

Здесь для объединения двух наборов используется библиотечный алгоритм `set_intersection`, описанный в приложении А.2.8.

Алгоритм `set_intersection` получает пять итераторов. Первые четыре он использует для обозначения двух исходных последовательностей (см. раздел 10.5.2). Его последний аргумент обозначает получателя. Алгоритм выводит элементы, присутствующие в обеих исходных последовательностях, в результирующую.

В данном вызове получателем является итератор вставки (см. раздел 10.4.1). Результатом записи алгоритмом `set_intersection` в этот итератор будет вставка нового элемента в набор `ret_lines`.

Подобно функции `eval()` класса `OrQuery`, эта завершается созданием и возвращением объекта класса `QueryResult`, представляющего объединение соответствий.

Функция `NotQuery::eval()`

Функция eval() класса NotQuery ищет в тексте все строки, в которых operand отсутствует.

```
// возвращает строки, отсутствующие в наборе
результатов
// operand QueryResult
NotQuery::eval(const TextQuery& text) const {
    // виртуальный вызов для вычисления operand'a Query
    auto result = query.eval(text);
    // начать с пустого результирующего набора данных
    auto ret_lines = make_shared<set<line_no>>();
    // следует перебрать строки, в которых
    // присутствует operand
    auto beg = result.begin(), end = result.end();
    // для каждой строки во входном файле, если она
    // отсутствует
    // в result, добавить ее номер в ret_lines
    auto sz = result.get_file()->size();
    for (size_t n = 0; n != sz; ++n) {
        // если не обработаны все строки в result
        // проверить присутствие этой строки
        if (beg == end || *beg != n)
            ret_lines->insert(n); // если нет в result,
        // добавить строку
        else if (beg != end)
            ++beg; // в противном случае получить следующий
        // номер строки
            // в result, если она есть
    }
    return QueryResult(rep(), ret_lines,
result.get_file());
}
```

Как и другие функции eval(), данная начинается с вызова функции eval() operand'a объекта. Этот вызов возвращает объект класса QueryResult, содержащий номера строк, в которых присутствует operand. Однако вернуть необходимо набор номеров строк, в которых operand отсутствует. Как и в других функциях eval(), данная начинается с вызова функции eval() operand'a объекта. Вызов возвращает объект класса QueryResult, содержащий номера строк, в которых operand

присутствует, но необходимы номера строки, на которых операнд отсутствует. Поэтому следует найти в файле все строки, отсутствующие в наборе результатов.

Набор создается в результате последовательного перебора целых чисел до размера входного файла. Каждое число, отсутствующее в наборе `result`, помещается в набор `ret_lines`. Итераторы `beg` и `end` устанавливаются на первый и следующий после последнего элементы в наборе `result`. Поскольку речь идет о наборе, при переборе номера строк будут следовать в порядке возрастания.

Тело цикла проверяет наличие текущего числа в наборе `result`. Если его нет, то число добавляется в набор `ret_lines`. Если он есть, осуществляется приращение итератора `beg` набора `result`.

Как только все номера строк будут обработаны, возвращается объект класса `QueryResult`, содержащий набор `ret_lines` наряду с результатами выполнения функций `rep()` и `get_file()`, как и у предыдущих функций `eval()`.

Упражнения раздела 15.9.4

Упражнение 15.39. Реализуйте классы `Query` и `Query_base`. Проверьте приложение на вычислении и выводе запроса, представленного на рис. 15.3.

Упражнение 15.40. Что будет, если параметр `rhs` функции-члена `eval()` класса `OrQuery` возвратит пустой набор? Что, если так поступит ее параметр `lhs`? Что если и `rhs`, и `lhs` возвратят пустые множества?

Упражнение 15.41. Переделайте свои классы так, чтобы использовать встроенные указатели на класс `Query_base`, а не интеллектуальные указатели `shared_ptr`. Помните, что ваши классы больше не смогут использовать синтезируемые функции-члены управления копированием.

Упражнение 15.42. Разработайте и реализуйте одно из следующих дополнений.

(а) Организуйте вывод слов только однажды в предложении, а не однажды в строке.

(б) Снабдите систему историей, позволяющей пользователю обратиться к предыдущему запросу по номеру, а также добавлять или комбинировать их с другими.

(с) Позвольте пользователю ограничивать результаты так, чтобы отображался набор соответствий только в заданном диапазоне строк.

Резюме

Наследование позволяет создавать новые классы, которые совместно используют возможности их базового класса (классов), но при необходимости могут их переопределить или дополнить. Динамическое связывание позволяет компилятору во время выполнения выбрать версию применяемой функции на основании динамического типа объекта. Комбинация наследования и динамического связывания позволяет создавать программы, которые либо не зависят от типа объекта, либо имеют поведение, зависящее от типа объекта.

В языке C++ динамическое связывание применимо *только* к тем функциям, которые объявлены виртуальными и вызываются при помощи ссылок или указателей.

Объекты производных классов состоят из части (частей) базового класса и части производного. Поскольку частью объекта производного класса является объект базового, ссылку или указатель на объект производного класса вполне можно преобразовать в ссылку или указатель на его доступный базовый класс.

При создании, копировании, перемещении и присвоении объектов производного класса сначала создается, копируется, перемещается и присваивается базовая часть объекта. Деструкторы выполняются в обратном порядке: сначала удаляется производная часть, затем выполняются деструкторы частей базовых классов. Базовые классы обычно определяют виртуальный деструктор, даже если у них нет никакой потребности в деструкторе.

Производный класс определяет уровень защиты для каждого из своих базовых классов. Члены открытого базового класса являются частью интерфейса производного класса; члены закрытого базового класса недоступны; члены защищенного базового класса доступны для классов, производных от него, но не для пользователей производного класса.

Термины

Абстрактный класс (abstract base class). Класс, обладающий одной или несколькими чистыми виртуальными функциями. Нельзя создать объекты типа абстрактного базового класса.

Базовый класс (base class). Класс, от которого происходит другой класс. Члены базового класса становятся членами производного класса.

Виртуальная функция (virtual function). Функция-член,

обеспечивающая зависимое от типа поведение. Во время выполнения выбор конкретной версии функции при обращении к виртуальной функции с помощью ссылки или указателя осуществляется на основании типа объекта, с которым связана ссылка или указатель.

Динамический тип (dynamic type). Тип объекта во время выполнения. Динамический тип объекта, на который ссылается ссылка или указывает указатель, может отличаться от статического типа ссылки или указателя. Указатель или ссылка на тип базового класса может применяться к объекту производного типа. В таких случаях статическим типом будет ссылка (или указатель) на базовый класс, а динамическим — ссылка (или указатель) на производный.

Динамическое связывание (dynamic binding). Отсрочка выбора выполняемой функции до времени выполнения. В языке C++ динамическим связыванием называют выбор во время выполнения используемой версии виртуальной функции на основании фактического типа объекта, который связан со ссылкой или с указателем.

Доступность (accessible). Член базового класса доступен через производный объект. Доступность зависит от спецификатора доступа, используемого в списке наследования производного класса, и уровня доступа члена в базовом классе. Например, открытый (`public`) член класса, унаследованный при открытом наследовании, доступен для пользователей производного класса. Открытый член базового класса недоступен, если наследование является закрытым.

Закрытое наследование (private inheritance). При закрытом наследовании открытые и защищенные члены базового класса становятся закрытыми членами производного.

Защищенное наследование (protected inheritance). При защищенном наследовании защищенные и открытые члены базового класса становятся защищенными членами производного.

Косвенный базовый класс (indirect base class). Базовый класс, отсутствующий в списке наследования производного класса. Класс, от которого наследуется прямой базовый класс, прямо или косвенно является косвенным базовым классом для производного класса.

Наследование (inheritance). Программная технология определения нового класса (производного) в терминах существующего класса (базового). Производный класс наследует члены базового класса.

Объектно-ориентированное программирование (object-oriented programming). Техника программирования с использованием абстракции данных, наследования и динамического связывания.

Открытое наследование (public inheritance). Открытый интерфейс базового класса является частью открытого интерфейса производного класса.

Отсечение (sliced down). Происходящее при использовании объекта производного типа для инициализации или присвоения объекта базового типа. Производная часть объекта отсекается, оставляя только базовую часть, которая и присваивается объекту базового типа.

Переопределение (override). Виртуальная функция, определенная в производном классе, с тем же списком параметров, что и у виртуальной функции в базовом классе, переопределяет определение базового класса.

Полиморфизм (polymorphism). Применительно к объектно-ориентированному программированию — возможность получить специфическое для типа поведение на основании динамического типа ссылки или указателя.

Преобразование производного класса в базовый (derived-to-base conversion). Неявное преобразование объекта производного класса в ссылку на базовый класс или указателя на объект производного класса в указатель на базовый класс.

Привязка во время выполнения (run-time binding). См. динамическое связывание.

Производный класс (derived class). Класс, унаследованный от другого класса. Производный класс может переопределить виртуальные функции своего базового класса и определять новые члены. Область видимости производного класса вкладывается в область ее базового класса (классов); члены производного класса могут использовать члены базового класса непосредственно.

Прямой базовый класс (direct base class). Базовый класс, от которого непосредственно происходит производный. Прямые базовые классы определяются в списке наследования производного класса. Прямой базовый класс сам может быть производным классом.

Рефакторинг (refactoring). Способ перепроектирования программ, позволяющий собрать взаимосвязанные части в единую абстракцию при замене первоначального кода новой абстракцией. Рефакторинг классов, как правило, применяют для перемещения переменных или функций-членов в самый верхний общий пункт иерархии во избежание дублирования кода.

Спецификатор доступа `protected`. К членам, определенным после ключевого слова `protected`, могут обращаться только члены производного класса и друзья. Однако доступны эти члены только через производные объекты. Защищенные члены не доступны для обычных

пользователей класса.

Список наследования класса (class derivation list). Список базовых классов, от которых происходит производный класс; у каждого из них может быть необязательный уровень доступа. Если спецификатора доступа нет, наследование открытое (`public`), если производный класс определен с ключевым словом `struct`, и закрытое (`private`), если класс определен с ключевым словом `class`.

Статический тип (static type). Тип, с которым определяется переменная или возвращает выражение. Статический тип известен во время компиляции.

Чистая виртуальная функция (pure virtual). Виртуальная функция, объявленная в заголовке класса с использованием = 0 в конце списка параметров функции. Чистая виртуальная функция не обязана (но вполне может) быть определена классом. Класс с чистой виртуальной функцией является абстрактным. Если производный класс не определяет собственную версию унаследованной чистой виртуальной функции, он также становится абстрактным.

Глава 16

Шаблоны и обобщенное программирование

И объектно-ориентированное, и обобщенное программирование имеют дело с типами, неизвестными на момент написания программы. Различие между ними в том, что объектно-ориентированное программирование имеет дело с типами, которые не известны до времени выполнения, тогда как в обобщенном программировании типы становятся известны только во время компиляции.

Все описанные в части II контейнеры, итераторы и алгоритмы являются хорошими примерами обобщенного программирования. При написании обобщенной программы ее код должен работать способом, независимым от специфических типов. При использовании обобщенного кода ему следует предоставить типы или значения, с которыми будет работать данный конкретный экземпляр кода.

Например, библиотека предоставляет единое, обобщенное определение каждого контейнера, такого как вектор. Это обобщенное определение можно использовать для определения множества разных типов векторов, каждый из которых отличается от других типом хранимых элементов.

Шаблоны (*template*) — это основа обобщенного программирования. Шаблоны вполне можно использовать (как выше в книге), даже не понимая, как они определяются. В этой главе рассматривается определение собственных шаблонов.

В языке C++ шаблоны являются основой для общего программирования. Шаблон — это проект или формула для создания класса или функции.

При использовании такого обобщенного типа, как `vector`, или такой обобщенной функции, как `find()`, следует предоставить дополнительную информацию, необходимую для трансформации их проекта в конкретный класс или функцию во время компиляции. Использование шаблонов рассматривалось в главе 3, а в этой главе мы изучим их определение.

16.1. Определение шаблона

Предположим, необходимо написать функцию, которая сравнивает два значения и указывает, является ли первое из них меньшим, равным или большим, чем второе. Фактически придется создать несколько таких функций, каждая из которых сможет сравнивать значения определенного типа. На первом этапе можно было бы определить несколько перегруженных функций.

```
// возвращает 0, если значения равны, -1, если v1
меньше, и 1,
// если меньше v2
int compare(const string &v1, const string &v2) {
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
int compare(const double &v1, const double &v2) {
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

Эти функции почти идентичны и отличаются только типом параметров. Тела у обеих функций одинаковы.

Повторение тела функции для каждого сравниваемого типа не только утомительно, но и повышает вероятность возникновения ошибок. Однако важней всего то, что в этом случае необходимо *заранее* знать все типы, которые придется сравнивать. Этот подход не сработает в случае, когда функцию предполагается использовать для типов, неизвестных на данный момент.



16.1.1. Шаблоны функций

Вместо того чтобы определять новую функцию для каждого типа, мы можем определить *шаблон функции* (function template). Шаблон функции — это проект, по которому можно создать некую версию данной функции, специфическую для заданного типа. Шаблон функции `compare()` может выглядеть так:

```
template <typename T>
int compare( const T &v1, const T &v2) {
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

Определение шаблона начинается с ключевого слова `template`, за которым следует разделяемый запятыми и заключенный в угловые скобки (`<>`) *список параметров шаблона* (template parameter list), один или несколько *параметров шаблона* (template parameter).



Список параметров в определении шаблона не может быть пустым

Список параметров шаблона очень похож на список параметров функции. Список параметров функции задает имена и типы локальных переменных, но оставляет их неинициализированными. Инициализацию параметров во время выполнения обеспечивают аргументы.

Аналогично параметры шаблона представляют типы или значения, используемые при определении класса или функции. При использовании шаблона необходимо (явно или неявно) определить *аргументы шаблона* (template argument), чтобы связать их с соответствующими параметрами шаблона.

Например, рассматриваемая функция `compare()` объявляет единственный параметр типа `T`. В шаблоне `compare` имя `T` можно использовать там, где должно быть название типа данных. *Фактический*

тип *T* будет определен компилятором на основании способа применения функции.

Создание экземпляра шаблона функции

Когда происходит вызов шаблона функции, для вывода типов аргументов шаблона компилятор обычно использует аргументы вызова. Таким образом, когда происходит вызов шаблона *compare*, компилятор использует тип аргументов для определения типа, связанного с параметром шаблона *T*. Рассмотрим следующий вызов:

```
cout << compare(1, 0) << endl; // T - тип int
```

Здесь аргумент имеет тип *int*. Компилятор выведет и использует тип *int* как аргумент шаблона, а также свяжет этот аргумент с параметром *T* шаблона.

При *создании экземпляра* (*instantiation*) специфической версии функции компилятор сам использует выведенные параметры шаблона. При этом он подставляет фактические аргументы шаблона вместо соответствующих параметров шаблона. Рассмотрим следующий вызов:

```
// создание экземпляра int compare(const int&, const int&)
cout << compare(1, 0) << endl; // T - тип int
// создание
// экземпляра int compare(const vector<int>&, const vector<int>&)
vector<int> vec1{1, 2, 3}, vec2{4, 5, 6};
cout << compare(vec1, vec2) << endl; // T - тип vector<int>
```

Здесь компилятор создает два экземпляра разных версий функции *compare()*. В первой из них параметр *T* заменен типом *int*.

```
int compare(const int &v1, const int &v2) {
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

Во втором вызове создается версия функции *compare()* с параметром *T*, замененным типом *vector<int>*. Такое создание компилятором функций обычно и называют *созданием экземпляра* шаблона.

Параметры типа шаблона

У функции *compare()* есть один *параметр типа* (*type parameter*)

шаблона. Как правило, параметр типа можно использовать как спецификатор типа таким же образом, как и встроенный спецификатор типа или класса. В частности, параметр типа применим при назначении типа возвращаемого значения или типа параметра функции, а также в объявлениях переменных или приведениях в теле функции:

```
// ok: для возвращаемого значения и параметра
используется тот же тип
template <typename T> T foo(T* p) {
    T tmp = *p; // тип tmp совпадает с типом, на
который указывает p
    // ...
    return tmp;
}
```

Каждому параметру типа должно предшествовать ключевое слово `class` или `typename`:

```
// ошибка: U должно предшествовать либо typename,
либо class
template <typename T, U> T calc(const T&, const
U&);
```

В списке параметров шаблона эти ключевые слова имеют одинаковый смысл и применяются взаимозаменяюще. Оба ключевых слова применимы одновременно:

```
// ok: в списке параметров шаблона нет никакой
разницы между ключевыми
// словами typename и class
template <typename T, class U> calc(const T&, const
U&);
```

Для обозначения параметра типа шаблона интуитивно понятней использовать ключевое слово `typename`, а не `class`; в конце концов, для фактического типа параметра вполне может быть использован встроенный тип, а не только класс. Кроме того, ключевое слово `typename` более точно указывает на то, что следующее за ним имя принадлежит типу. Однако ключевое слово `typename` было добавлено в язык C++ как часть стандарта C++, поэтому в устаревших программах, вероятнее всего, осталось исключительно ключевое слово `class`.

Параметры значения шаблона

Кроме параметров типа, в определении шаблона могут быть использованы *параметры значения* (*nontype parameter*). Параметр значения

представляет значение, а не тип. При определении параметров значения вместо ключевого слова `class` или `typename` используются имена типов.

При создании экземпляра шаблона такие параметры заменяются значением, предоставленным пользователем или выведенным компилятором. Чтобы компилятор смог создать экземпляр шаблона во время компиляции, эти значения должны быть константными выражениями (см. раздел 2.4.4).

В качестве примера напишем версию функции `compare()`, работающую со строковыми литералами. Такие литералы представляют собой массивы типа `const char`. Поскольку скопировать массив нельзя, определим параметры как ссылки на массив (раздел 6.2.4). Поскольку необходима возможность сравнивать литералы разных длин, снабдим шаблон двумя параметрами значения. Первый параметр шаблона представляет размер первого массива, а второй — размер второго:

```
template<unsigned N, unsigned M>
int compare(const char (&p1)[N], const char (&p2)
[M]) {
    return strcmp(p1, p2);
}
```

При вызове следующей версии функции `compare()` компилятор будет использовать размер литералов для создания экземпляра шаблона с размерами, которыми заменяют параметры `N` и `M`:

```
compare("hi", "mom")
```

Не забывайте, что компилятор завершает строковый литерал пустым символом (см. раздел 2.1.3). В результате компилятор создаст такой экземпляр:

```
int compare(const char (&p1)[3], const char (&p2)
[4])
```

Параметр значения может быть целочисленным типом, указателем, ссылкой на объект (l-значением) или на тип функции. Аргумент, связанный с целочисленным параметром значения, должен быть константным выражением. У аргументов, привязанных к указателю или ссылочному параметру значения, должна быть статическая продолжительность существования (см. главу 12). Нельзя использовать обычный (нестатический) локальный или динамический объект как аргумент шаблона для параметра значения шаблона в виде ссылки или указателя. Параметр-указатель может быть также создан как `nullptr` или нулевое константное выражение.

Параметр значения шаблона — это константное значение в определении шаблона. Параметр значения применим там, где требуются константные выражения, например, при определении размера массива.



Аргументы шаблона, используемые для параметров значения, должны быть константными выражениями.

Шаблоны функции со спецификаторами `inline` и `constexpr`

Шаблон функции может быть объявлен как `inline` (встраиваемый) или `constexpr`, как и обычная функция. Спецификаторы `inline` и `constexpr` располагаются после списка параметров шаблона, но перед типом возвращаемого значения.

```
// ok: спецификатор inline следует за списком
параметров шаблона
```

```
template <typename T> inline T min(const T&, const
T&);
```

```
// ошибка: неправильное размещение спецификатора
inline
```

```
inline template <typename T> T min(const T&, const
T&);
```



Создание кода, независимого от типа

Продемонстрируем два наиболее важных принципа создания обобщенного кода на примере функции `compare()`.

- Параметры функций в шаблоне должны быть ссылками на константу.
- При проверке в теле шаблона следует использовать только оператор сравнения `<`.

Объявление параметров функций ссылками на константы гарантирует возможность применения функции к типам, которые не допускают копирования. Большинство типов, включая встроенные типы, но исключая указатели `unique_ptr` и типы ввода-вывода, а также все использованные ранее библиотечные типы допускают копирование. Но вполне могут

встретиться и другие типы, которые не допускают копирования. Сделав параметры ссылками на константы, можно гарантировать применимость таких типов в функции `compare()`. Кроме того, если функция `compare()` будет применена для больших объектов, такая конструкция позволит избежать копирования и сэкономит время при выполнении.

Некоторые читатели могут подумать, что для сравнения было бы целесообразней использовать оба оператора `<` и `>`.

```
// ожидаемое сравнение
if (v1 < v2) return -1;
if (v1 > v2) return 1;
return 0;
```

Однако написание кода, использующего только оператор `<`, снизит требования к типам, которые применимы в функции `compare()`. Эти типы должны поддерживать оператор `<`, но не обязаны поддерживать оператор `>`.

Фактически, если действительно следует обеспечить независимость от типа и переносимость кода, лучше определить свою функцию, используя тип `less` (см. раздел 14.8.2):

```
// версия функции compare(), корректно работающая
даже с
// указателями; см. р. 14.8.2
template <typename T> int compare(const T &v1,
const T &v2) {
    if (less<T>()(v1, v2)) return -1;
    if (less<T>()(v2, v1)) return 1;
    return 0;
}
```

Проблема первоначальной версии в том, что если пользователь вызовет ее с двумя указателями, не указывающими на тот же массив, то результат выполнения кода будет непредсказуем.

Рекомендуем

При написании кода шаблонов следует постараться минимизировать количество требований, накладываемых на типы аргументов.



Компиляция шаблона

Когда компилятор встречает определение шаблона, он не создает код. Код создается только при создании специфического экземпляра шаблона. Тот факт, что код создается только при использовании шаблона (а не при его определении), влияет как на организацию исходного кода, так и на способы обнаружения ошибок.

Обычно, когда происходит вызов функции, компилятору достаточно объявления функции. Точно так же при использовании объекта класса должно быть доступно определение класса, но определения функций-членов не обязательны. В результате определения классов и объявления функций имеет смысл размещать в файлах заголовка, а определения обычных функций и функций-членов — в файлах исходного кода.

С шаблонами все не так: для создания экземпляра у компилятора должен быть код, определяющий шаблон функции или функцию-член шаблона класса. В результате, в отличие от обычного кода, заголовки для шаблонов обычно включают определения наравне с объявлениями.



Определения шаблонов функций и функций-членов шаблонов классов обычно помещаются в файлы заголовка.

Ключевая концепция. Шаблоны и заголовки

Шаблоны содержат два вида имен:

- не зависящие от параметров шаблона;
- зависящие от параметров шаблона.

Именно разработчик шаблона гарантирует, что все имена, не зависящие от параметров шаблона, будут видимы на момент использования шаблона. Кроме того, разработчик шаблона должен гарантировать видимость определения шаблона, включая определения членов шаблона класса, на момент создания экземпляра шаблона.

Пользователь шаблона должен обеспечить видимость объявлений всех функций, типов и связанных с ними операторов, используемых при создании экземпляра шаблона.

Выполнение этих требований невозможно без хорошо организованной структуры программы, в которой заголовки используются соответствующим образом. Автор шаблона должен

предоставить заголовок, который содержит объявления всех имен, используемых в шаблоне класса или в определениях его членов. Прежде чем создать экземпляр шаблона для определенного типа или использовать член класса, созданного по этому шаблону, пользователь должен подключить заголовок для типа шаблона и заголовок, в котором определен используемый тип.

Ошибки компиляции проявляются, главным образом, во время создания экземпляра

Тот факт, что код не создается до создания экземпляра шаблона, влияет на то, когда проявляются ошибки компиляции в коде шаблона. В процессе создания шаблона есть три этапа, во время которых компилятор может сообщить об ошибке.

Первый — когда компилируется само определение шаблона. На этом этапе компилятор, как правило, не может найти большую часть ошибок. Здесь обнаруживаются в основном синтаксические ошибки, такие как пропущенная точка с запятой или неправильно написанное имя переменной, но не более.

Второй этап обнаружения ошибок — когда компилятор встречает применение шаблона. На данном этапе компилятор также способен проверить немногое. Для вызова шаблона функции компилятор обычно проверяя количество и типы аргументов. Он может также проверить совпадение типов двух аргументов. Для шаблона класса компилятор может проверить количество и правильность предоставленных аргументов шаблона, но не более.

Третий этап обнаружения ошибок — момент создания экземпляра. Только теперь обнаруживаются ошибки, связанные с типами. В зависимости от того, как компилятор осуществляет создание экземпляра, он может сообщить об этих ошибках во время редактирования.

При написании шаблона код не может быть открыто специфическим для типа, но можно сделать некоторые предположения об используемых типах. Например, код первоначальной функции `comparage()` подразумевал, что тип аргумента имеет оператор `<`.

```
if (v1 < v2) return -1; // для объектов типа T
требуется оператор <
if (v2 < v1) return 1; // для объектов типа T
требуется оператор <
return 0; // возвращает int; не
зависит от T
```

Когда компилятор обрабатывает тело этого шаблона, он не может проверить корректность условий в операторах `if`. Если переданные функции `compare()` аргументы имеют оператор `<`, то код сработает прекрасно, но не в противном случае. Например:

```
Sales_data data1, data2;  
cout << compare(data1, data2) << endl; // ошибка: у  
Sales_data нет  
// оператора  
<
```

Этот вызов создает экземпляр функции `compare()` с параметром `T`, замененным классом `Sales_data`. Если условия попытаются использовать оператор `<` для объектов класса `Sales_data`, то окажется, что такого оператора нет. В результате получится экземпляр функции, которая не будет откомпилирована. Такие ошибки, как эта, не могут быть обнаружены, пока компилятор не создаст экземпляр определения функции `compare()` для типа `Sales_data`.



Вызывающая сторона должна гарантировать, что переданные шаблону аргументы поддерживают все используемые им операторы, а также то, что эти операторы будут вести себя правильно в том контексте, в котором шаблон использует их.

Упражнения раздела 16.1.1

Упражнение 16.1. Определите создание экземпляра.

Упражнение 16.2. Напишите и проверьте собственные версии функций `compare()`.

Упражнение 16.3. Вызовите собственную функцию `compare()` для объекта класса `Sales_data` и посмотрите, как ваш компилятор обрабатывает ошибки во время создания экземпляра.

Упражнение 16.4. Напишите шаблон, действующий как библиотечный алгоритм `find()`. Функция будет нуждаться в двух параметрах типа шаблона: один — для представления параметров-итераторов функции и другой — для типа значения. Используйте свою функцию для поиска заданного значение в векторе `vector<int>` и списке `list<string>`.

Упражнение 16.5. Напишите шаблон функции `print()` из раздела

6.2.4, которая получает ссылку на массив и может обрабатывать массивы любого размера и любого типа элементов.

Упражнение 16.6. Как работают библиотечные функции `begin()` и `end()`, получающие аргумент в виде массива? Определите собственные версии этих функций.

Упражнение 16.7. Напишите шаблон `constexpr`, возвращающий размер заданного массива.

Упражнение 16.8. В разделе "Ключевая концепция" в разделе 3.4.1 упоминалось о том, что программисты C++ привыкли использовать оператор `!=`, а не `<`. Объясните причину этой привычки.



16.1.2. Шаблоны класса

Шаблон класса (class template) — своего рода проект для создания классов. Шаблоны классов отличаются от шаблонов функций, для которых компилятор не может вывести типы параметров шаблона. Вместо этого, как уже демонстрировалось не раз, для использования шаблона класса следует предоставить дополнительную информацию в угловых скобках после имени шаблона (см. раздел 3.3). Эта дополнительная информация — список аргументов шаблона, подставляемых вместо параметров шаблона.

Определение шаблона класса

В качестве примера реализуем шаблонную версию класса `StrBlob` (см. раздел 12.1.1). Присвоим шаблону имя `Blob`, указывающее, что он больше не специфичен только для строк. Как и класс `StrBlob`, этот шаблон будет предоставлять совместный (и проверяемый) доступ к своим членам. В отличие от класса, шаблон применяется к элементам практически любого типа. Подобно библиотечным контейнерам, используя шаблон `Blob`, пользователи должны будут определить тип элемента.

Как и шаблоны функции, шаблоны класса начинаются с ключевого слова `template`, за которым следует список параметров шаблона. В определении шаблона класса (и его членов) используются параметры шаблона как знакомства типов или значений, которые будут подставлены при использовании шаблона:

```
template <typename T> class Blob {
public:
    typedef T value_type;
    typedef      typename      std::vector<T>::size_type
size_type;
    // конструкторы
    Blob( );
    Blob( std::initializer_list<T> il );
    // количество элементов в Blob
    size_type size( ) const { return data->size( ); }
    bool empty( ) const { return data->empty( ); }
    // добавление и удаление элементов
```

```

void push_back( const T &t) { data->push_back( t); }
// версия перемещения; см. р. 13.6.3
void push_back( T &&t) { data-
>push_back( std::move( t)); }
void pop_back();
// доступ к элементу
T& back();
T& operator[]( size_type i); // определено в
разделе 14.5
private:
std::shared_ptr<std::vector<T>> data;
// выдать сообщение, если data[ i] недопустим
void check( size_type i, const std::string &msg)
const;
}

```

У шаблона `Blob` есть один параметр типа `T`. Он используется везде, где ожидается тип элемента, хранимый классом `Blob`. Например, тип возвращаемого значения функции доступа к элементам `Blob` определен как `T&`. Когда пользователь создаст экземпляр шаблона `Blob`, он использует параметр `T` для замены конкретным типом аргумента шаблона.

За исключением списка параметров шаблона и использования `T` вместо `string`, этот класс совпадает с тем, что было определено в разделе 12.1.1 и модифицировано в разделе 12.1.6, а также в главах 13 и 14.

Создание экземпляра шаблона класса

Как уже неоднократно упоминалось, при использовании шаблона класса следует предоставить дополнительную информацию. Как можно теперь утверждать, эта дополнительная информация является списком явных аргументов шаблона (explicit template argument), которые привязаны к параметрам шаблона. Компилятор использует эти аргументы для создания специфического экземпляра класса по шаблону.

Например, чтобы определить тип для шаблона `Blob`, следует предоставить тип элемента:

```

Blob<int> ia; // пустой Blob<int>
Blob<int> ia2 = { 0,1,2,3,4}; // Blob<int> с пятью
элементами

```

Оба объекта, `ia` и `ia2`, используют ту же специфическую для типа версию шаблона `Blob` (т.е. `Blob<int>`). Из этих определений

компилятор создает экземпляр класса, который эквивалентен следующему:

```
template <> class Blob<int> {
    typedef typename std::vector<int>::size_type
size_type;
    Blob();
    Blob(std::initializer_list<int> il);
    // ...
    int& operator[](size_type i);
private:
    std::shared_ptr<std::vector<int>> data;
    void check(size_type i, const std::string &msg)
const;
};
```

Когда компилятор создает экземпляр класса из шаблона `Blob`, он переписывает его, заменяя каждый экземпляр параметра `T` заданным аргументом шаблона, которым в данном случае является `int`.

Компилятор создает разный класс для каждого заданного типа элемента:

```
// эти определения создают экземпляр двух разных
типов Blob
Blob<string> names; // Blob содержащий строки
Blob<double> prices; // другой тип элемента
```

Эти определения привели бы к созданию двух разных экземпляров класса: определение `names` создает класс `Blob`, в котором каждое вхождение `T` заменено на `string`. Определение `prices` создает класс `Blob`, где `T` заменено на `double`.



При каждом создании экземпляра шаблона класса получается независимый класс. У типа `Blob<string>` нет никаких отношений с другим типом класса `Blob` или специальных прав доступа к его членам.



Ссылки на тип шаблона в пределах шаблона

При чтении кода шаблона класса не следует забывать, что имя шаблона

класса не является именем самого класса (см. раздел 3.3). Шаблон класса используется для создания экземпляра класса, при этом всегда используются аргументы шаблона.

Непонятным может показаться то, что код в шаблоне класса вообще не использует имя фактического типа (или значения) как аргумент шаблона. Вместо этого как аргументы шаблона зачастую используются собственные параметры. Например, переменная-член `data` использует два шаблона, `vector` и `shared_ptr`. Каждый раз, когда используется шаблон, следует предоставить аргументы шаблона. В данном случае предоставляемый аргумент шаблона имеет тот же тип, который используется при создании экземпляра шаблона `Blob`. Следовательно, определение переменной-члена `data` с использованием параметра типа шаблона `Blob` свидетельствует о том, что переменная-член `data` является экземпляром указателя `shared_ptr` на экземпляр шаблона `vector`, содержащего объекты типа `T`.

```
std::shared_ptr<std::vector<T>> data;
```

При создании экземпляра специфического класса `Blob`, такого как `Blob<string>`, переменная-член `data` будет такой:

```
shared_ptr<vector<string>>
```

Если создать экземпляр `Blob<int>`, то переменная-член `data` будет такой: `shared_ptr<vector<int>>`, и т.д.

Функции-члены шаблонов класса

Подобно любому классу, функции-члены шаблона класса можно определить как в, так и вне тела класса. Как и у любых других классов, члены, определенные в теле, неявно являются встраиваемыми.

Функция-член шаблона класса сама по себе является обычной функцией. Однако у каждого экземпляра шаблона класса есть собственная версия каждого члена. В результате у функции-члена шаблона класса будут те же параметры шаблона, что и у самого класса. Поэтому функция-член, определенная вне тела шаблона класса, начинается с ключевого слова `template`, сопровождаемого списком параметров шаблона класса.

Как обычно, при определении члена класса вне его тела следует указать, к какому классу он принадлежит. Так же, как обычно, имя созданного из шаблона класса включает его аргументы шаблона. При определении члена аргументы шаблона совпадают с параметрами шаблона. Таким образом, для функции-члена класса `StrBlob`, определенной следующим образом:

тип_возвращаемого_значения
StrBlob::имя_члена(список_параметров)
соответствующий член шаблона Blob будет выглядеть так:
template <typename T>
тип_возвращаемого_значения
Blob<T>::имя_члена(список_параметров)

Функция *check()* и функции доступа к членам

Начнем с определения функции-члена *check()*, проверяющей предоставленный индекс:

```
template <typename T>
void Blob<T>::check(size_type i, const std::string
&msg) const {
    if (i >= data->size())
        throw std::out_of_range(msg);
}
```

Кроме отличия в имени класса и использовании списка параметров шаблона, эта функция идентична первоначальной функции-члену класса StrBlob.

Оператор индексирования и функция *back()* используют параметр шаблона для определения типа возвращаемого значения, но в остальном они неизменны:

```
template <typename T>
T& Blob<T>::back() {
    check(0, "back on empty Blob");
    return data->back();
}

template <typename T>
T& Blob<T>::operator[](size_type i) {
    // если i слишком велико, check() передаст
    // сообщение и предотвратит
    // доступ к несуществующему элементу
    check(i, "subscript out of range");
    return (*data)[i];
}
```

В первоначальном классе StrBlob эти операторы возвращали тип *string&*. Шаблонная версия возвращает ссылку на любой тип, использованный при создании экземпляра шаблона Blob.

Функция `pop_back()` почти идентична оригинальной функции-члену класса `StrBlob`:

```
template <typename T> void Blob<T>::pop_back() {
    check(0, "pop_back on empty Blob");
    data->pop_back();
}
```

Оператор индексирования и функция-член `back()` перегружены как `const`. Оставим определение этих функций-членов и функции `front()` читателю в качестве самостоятельного упражнения.

Конструкторы `Blob`)

Подобно любым другим функциям-членам, определенным вне шаблона класса, конструктор начинается с объявления параметров шаблона для шаблона класса, членом которого он является:

```
template <typename T>
Blob<T>::Blob():
data(std::make_shared<std::vector<T>>()) { }
```

Здесь функция-член `Blob()` определяется в пределах шаблона `Blob<T>`. Как и стандартный конструктор `StrBlob()` (см. раздел 12.1.1), данный конструктор резервирует пустой вектор и сохраняет указатель на него в переменной `data`. Как уже упоминалось, в качестве аргумента резервируемого шаблона `vector` используется собственный параметр типа класса.

Точно так же конструктор, получающий параметр типа `initializer_list`, использует свой параметр типа `T` как тип элемента для своего параметра типа `initializer_list`:

```
template <typename T>
Blob<T>::Blob(std::initializer_list<T> il):
data(std::make_shared<std::vector<T>>(il)) { }
```

Подобно стандартному конструктору, этот конструктор резервирует новый вектор. В данном случае этот вектор инициализируется из параметра `il`.

Чтобы использовать этот конструктор, следует передать список инициализации, тип элементов которого совместим с типом элемента `Blob`:

```
Blob<string> articles = {"a", "an", "the"};
```

Параметр этого конструктора имеет тип `initializer_list<string>`. Каждый строковый литерал в списке

неявно преобразуется в тип `string`.

Создание функций-членов шаблона класса

По умолчанию экземпляр функции-члена шаблона класса создается, только если программа использует эту функцию-член. Рассмотрим следующий код:

```
// создает экземпляр Blob<int> и конструктор
initializer_list<int>
Blob<int> squares = {0,1,2,3,4,5,6,7,8,9};
// создает экземпляр Blob<int>::size() const
for (size_t i = 0; i != squares.size(); ++i)
    squares[i] = i*i; // создает экземпляр
Blob<int>::operator[](size_t)
```

Этот код создает экземпляр класса `Blob<int>` и трех его функций-членов: `operator[]()`, `size()` и конструктора `initializer_list<int>()`.

Если функция-член не используется, ее экземпляр не создается. Благодаря этому факту можно создавать экземпляры класса, используя типы, которые не отвечают требованиям для некоторых из операций шаблона (см. раздел 9.2).



По умолчанию экземпляр члена шаблона класса создается, только если он используется.

Упрощение использования имени шаблона класса в коде класса

Из правила, согласно которому следует предоставить аргументы шаблона при использовании шаблона класса, есть одно исключение. В области видимости самого шаблона класса имя шаблона можно использовать без аргументов:

```
// BlobPtr передает исключение при попытке доступа
к несуществующему
// элементу
template <typename T> class BlobPtr
public:
    BlobPtr(): curr(0) { }
```

```

BlobPtr( Blob<T> &a, size_t sz = 0):
    wptr( a.data), curr( sz) { } T& operator*() const {
        auto p = check{ curr, "dereference past end");
        return (*p)[curr]; // (*p) - вектор, на который
указывает этот
                                // объект
    }
    // инкремент и декремент
    BlobPtr& operator++(); // префиксные операторы
    BlobPtr& operator--();
private:
    // если проверка успешна, check() возвращает
shared_ptr на вектор
    std::shared_ptr<std::vector<T>>
        check( std::size_t, const std::string&) const;
    // хранит weak_ptr, а значит, базовый вектор может
быть удален
    std::weak_ptr<std::vector<T>> wptr;
    std::size_t curr; // текущая позиция в пределах
массива
};

```

Внимательные читатели, вероятно, обратили внимание на то, что префиксные функции-члены инкремента и декремента шаблона класса `BlobPtr` возвращают тип `BlobPtr&`, а не `BlobPtr<T>&`. В области видимости шаблона класса компилятор рассматривает ссылки на сам шаблон так, как будто были подставлены аргументы шаблона, соответствующие собственным параметрам. Таким образом, этот код эквивалентен следующему:

```

BlobPtr<T>& operator++();
BlobPtr<T>& operator--();

```

Использование имени шаблона класса вне тела шаблона

При определении функций-членов вне тела шаблона класса следует помнить, что код находится не в области видимости класса, пока не встретилось имя класса (см. раздел 7.4):

```

// постфикс: осуществляет инкремент/декремент
объекта, но возвращает
// неизменное значение
template <typename T>

```

```

BlobPtr<T> BlobPtr<T>::operator++( int ) {
    // никакой проверки здесь не нужно; ее выполнит
    // вызов префиксного
    // инкремента
    BlobPtr ret = *this; // сохранить текущее значение
    ++*this;           // перемещение на один
    // элемент; префиксный ++
    // проверяет инкремент
    return ret;        // возвратить сохраненное
    // состояние
}

```

Поскольку тип возвращаемого значения присутствует вне области видимости класса, следует указать, что он возвращает экземпляр `BlobPtr`, созданный с тем же типом, что и класс. В теле функции код находится в пределах класса, поэтому не нужно повторять аргумент шаблона при определении `ret`. Когда аргументы шаблона не предоставлены, компилятор подразумевает, что используется тот же тип, что и при создании экземпляра функции-члена. Следовательно, определение `ret` будет эквивалентно следующему:

```
BlobPtr<T> ret = *this;
```



В области видимости шаблона класса можно обращаться к шаблону, не определяя его аргументы.

Шаблоны классов и дружественные отношения

Когда класс объявляет дружественные отношения (см. раздел 7.2.1), класс и его друг могут быть или не быть шаблонами. Шаблон класса, у которого есть друг, не являющийся шаблоном, предоставляет дружественный доступ ко всем экземплярам шаблона. Когда друг сам является шаблоном, предоставляющий дружественные отношения класс контролирует, распространяются ли они на все экземпляры шаблона или только на некий специфический экземпляр.

Дружественные отношения "один к одному"

Наиболее распространенная форма дружественных отношений между

шаблоном класса и другим шаблоном (класса или функции) подразумевает дружбу между соответствующими экземплярами класса и его друга. Например, класс `Blob` должен объявить дружественным класс `BlobPtr` и шаблонную версию оператора равенства класса `Blob` (первоначально определенную для класса `StrBlob` в упражнении раздела 14.3.1).

Чтобы обратиться к определенному экземпляру шаблона (класса или функции), следует сначала объявить сам шаблон. Объявление шаблона включает список параметров шаблона:

```
// для объявления дружественных отношений в шаблоне
Blob нужны
// предварительные объявления
template <typename> class BlobPtr;
template <typename> class Blob; // необходимо для
параметров operator==
template <typename T>
bool operator==(const Blob<T>&, const Blob<T>&);
template <typename T> class Blob {
    // каждый экземпляр Blob предоставляет доступ к
версии BlobPtr и
    // оператору равенства экземпляра, созданного с
тем же типом
    friend class BlobPtr<T>;
    friend bool operator==<T>
        (const Blob<T>&, const Blob<T>&);
    // другие члены, как в разделе 12.1.1
};
```

Начнем с объявления `Blob`, `BlobPtr` и `operator==` шаблонами. Эти объявления необходимы для объявления параметра в функции `operator==` и дружественных объявлений в шаблоне `Blob`.

Объявления дружественными используют параметр шаблона `Blob` как собственный аргумент шаблона. Таким образом, дружба ограничивается этими экземплярами шаблона `BlobPtr` и оператора равенства, которые создаются с тем же типом:

```
Blob<char> ca; // BlobPtr<char> и operator==<char>
друзья
Blob<int> ia; // BlobPtr<int> и operator==<int>
друзья
```

Члены класса `BlobPtr<char>` могут обращаться к не открытым

Чтобы позволить создавать все экземпляры как дружественные, объявление дружественных отношений должно использовать параметры шаблона, которые отличаются от используемых самим классом.



Объявление параметра типа шаблона дружественным

По новому стандарту параметр типа шаблона можно сделать дружественным:

```
template <typename Type> class Bar {  
    friend Type; // предоставить доступ к типу,  
используемому для создания  
        // экземпляра Bar  
    // ...  
};
```

Здесь указано, что, независимо от используемого для создания экземпляра типа, класс `Bar` будет дружественным. Таким образом, для некоего типа под названием `Foo` он был бы другом для `Bar<Foo>`, а тип `Sales_data` — другом для `Bar<Sales_data>` и т.д.

Следует заметить, что хотя другом обычно бывает класс или функция, для класса `Bar` вполне допустимо создание экземпляра со встроенным типом. Такие дружественные отношения позволяют создавать экземпляры таких классов, как `Bar` со встроенными типами.

Псевдонимы типа шаблона

Экземпляр шаблона класса определяет тип класса, и, подобно любому другому типу класса, для экземпляра класса можно определить псевдоним при помощи ключевого слова `typedef` (см. раздел 2.5.1):

```
typedef Blob<string> StrBlob;
```

Это определение типа позволит выполнить код, написанный в разделе 12.1.1, используя текущую версию шаблона `Blob`, экземпляр которого создан для типа `string`. Поскольку шаблон не тип, ключевое слово `typedef` к шаблону неприменимо. Таким образом, нет никакого способа определить `typedef` для шаблона `Blob<T>`.



Однако новый стандарт позволяет определять псевдоним типа для шаблона класса:

```
template<typename T> using twin = pair<T, T>;
twin<string> authors; // authors - это pair<string,
string>
```

где имя `twin` определено как синоним для пар с одинаковыми типами членов. Пользователям типа `twin` достаточно определить его только однажды.

Псевдоним типа шаблона — это синоним для целого семейства классов:

```
twin<int> win_loss; // win_loss - это pair<int,
int>
twin<double> area; // area - это pair<double,
double>
```

Как и при использовании шаблона класса, при использовании псевдонима `twin` следует указать, какой именно вид `twin` необходим.

При определении псевдонима типа шаблона можно зафиксировать один или несколько параметров шаблона:

```
template <typename T> using partNo = pair<T,
unsigned>;
partNo<string> books; // books - это pair<string,
unsigned>
partNo<Vehicle> cars; // cars - это pair<Vehicle,
unsigned>
partNo<Student> kids; // kids - это pair<Student,
unsigned>
```

Здесь имя `partNo` определено как синоним семейства типов, которые являются парами, вторая переменная-член которого имеет тип `unsigned`. Пользователи `partNo` определяют тип первой переменной-члена пары, но не второй.

Статические члены шаблонов класса

Подобно любому другому классу, шаблон класса способен объявить статические члены (см. раздел 7.6):

```
template <typename T> class Foo {
public:
    static std::size_t count() { return ctr; }
    // другие члены интерфейса
private:
    static std::size_t ctr;
    // другие члены реализации
};
```

где `Foo` — шаблон класса, у которого есть открытая статическая функция-член `count()` и закрытая статическая переменная-член `ctr`. У каждого экземпляра шаблона `Foo` будет собственный экземпляр статических членов. Таким образом, для любого конкретного типа `X` будет по одной переменной `Foo<X>::ctr` и одной функции `Foo<X>::count()`. Все объекты типа `Foo<X>` будут совместно использовать ту же переименованную `ctr` и функцию `count()`. Например:

```
// создает экземпляр статических членов
Foo<string>::ctr
// и Foo<string>::count
Foo<string> fs;
// все три объекта совместно используют те же члены
Foo<int>::ctr
// и Foo<int>::count
Foo<int> fi, fi2, fi3;
```

Подобно любой другой статической переменной-члену, у каждой статической переменной-члена шаблона класса должно быть только одно определение. Однако для каждого экземпляра шаблона класса будет отдельный объект. В результате статическую переменную-член шаблона определяют таким же образом, как и функции-члены этого шаблона:

```
template <typename T>
size_t Foo<T>::ctr = 0; // определение и
инициализация ctr
```

Подобно любым другим членам шаблона класса, начнем с определения списка параметров шаблона, сопровождаемого типом и именем определяемого члена. Как обычно, имя члена включает имя класса, которое включает для класса, созданного из шаблона, его аргументы шаблона. Таким образом, когда класс `Foo` создается как экземпляр для специфического типа аргумента шаблона, для этого класса будет создан отдельный экземпляр переменной `ctr` и инициализирован значением 0.

Подобно статическим членам обычного класса, к статическому члену шаблона класса можно обратиться через объект класса или непосредственно, при помощи оператора области видимости. Конечно, чтобы использовать статический член через класс, следует обратиться к его конкретному экземпляру:

```
Foo<int> fi; // создает экземпляр
класса Foo<int> // и статической
```

переменной-члена *ctr*

```

auto ct = Foo<int>::count(); // создает экземпляр
Foo<int>::count()
ct = fi.count(); // использует
Foo<int>::count()
ct = Foo::count(); // ошибка: экземпляр
какого именно // шаблона создается?
```

Как и любая другая функция-член, экземпляр статической функции-члена создается только при его использовании в программе.

Упражнения раздела 16.1.2

Упражнение 16.9. Что такое шаблон функции? Что такое шаблон класса?

Упражнение 16.10. Что происходит при создании экземпляра шаблона класса?

Упражнение 16.11. Следующее определение шаблона *List* неправильно. Как его исправить?

```

template <typename elemType> class ListItem;
template <typename elemType> class List {
public:
    List<elemType>();
    List<elemType>( const List<elemType> & );
    List<elemType>& operator=( const List<elemType> & );
    ~List();
    void insert( ListItem *ptr, elemType value );
private:
    ListItem *front, *end;
};
```

Упражнение 16.12. Напишите собственные версии шаблонов *Blob* и *BlobPtr*, включая все константные члены, которые не были представлены в тексте.

Упражнение 16.13. Объясните, какой вид дружественных отношений вы выбрали бы для операторов равенства и сравнения шаблона *BlobPtr*.

Упражнение 16.14. Напишите шаблон класса *Screen*, который использует параметры значения для определения высоты и ширины экрана.

Упражнение 16.15. Реализуйте операторы ввода и вывода для своего шаблона *Screen*. Какие друзья необходимы классу *Screen* (если таковые вообще имеются) для работы операторов ввода и вывода? Объясните,

зачем нужно каждое объявление дружественным (если таковые вообще имеются).

Упражнение 16.16. Перепишите класс `StrVec` (см. раздел 13.5), как шаблон `Vec`.



16.1.3. Параметры шаблона

Подобно именам параметров функций, имена параметров шаблона не имеют никакого значения. Обычно параметрам типа присваивают имя `T`, но можно использовать любое другое:

```
template <typename Foo> Foo calc(const Foo& a,  
const Foo& b) {  
    Foo tmp = a; // тип tmp совпадает с типом  
    // параметров и возвращаемого  
    // значения  
    // ...  
    return tmp; // типы возвращаемого значения и  
    // параметров совпадают  
}
```

Параметры шаблона и область видимости

Параметры шаблона следуют обычным правилам области видимости. Имя параметра шаблона применимо сразу после его объявления и до конца объявления или определения шаблона. Подобно любым другим именам, параметр шаблона скрывает любые объявления имен во внешней области видимости. Однако, в отличие от большинства других контекстов, имя, используемое как параметр шаблона, не может быть повторно использовано в пределах шаблона:

```
typedef double A;  
template <typename A, typename B> void f( A a, B b)  
{  
    A tmp = a; // tmp имеет тип параметра шаблона A, а  
    // не double  
    double B; // ошибка: повторное объявление  
    // параметра шаблона B  
}
```

Согласно обычным правилам скрытия имен, определение `typedef` типа `A` скрывается определением параметра типа по имени `A`. Таким образом, переменная `tmp` не будет иметь тип `double`; она будет иметь любой тип, который будет передан параметру шаблона `A` при

использовании шаблона. Поскольку нельзя многократно использовать имена параметров шаблона, объявление переменной по имени *V* ошибочно.

Поскольку имя параметра не может быть использовано многократно, в каждом списке параметров шаблона имя параметра шаблона может присутствовать только однажды:

```
// ошибка: повторение имени V в параметрах шаблона
недопустимо
```

```
template <typename V, typename V> // ...
```

Объявления шаблона

Объявление шаблона должно включить параметры шаблона:

```
// объявляет, но не определяет compare и Blob
template <typename T> int compare(const T&, const
T&);
```

```
template <typename T> class Blob;
```

Подобно параметрам функций, имена параметров шаблона не должны совпадать с таковыми в объявлениях и определениях того же шаблона:

```
// все три случая использования calc
// относятся к тому же шаблону функции
template <typename T> T calc(const T&, const T&);
// объявление
template <typename U> U calc(const U&, const U&);
// объявление
// определение шаблона
template <typename Type>
Type calc(const Type& a, const Type& b) { /* ... */ }
```

Конечно, у каждого объявления и определения шаблона должно быть то же количество и вид (т.е. тип или значение) параметров.

Рекомендуем

По причинам, рассматриваемым в разделе 16.3, объявления всех шаблонов, необходимых данному файлу, обычно располагаются вместе в начале файла перед любым использующим их кодом.

Использование членов типа

Помните, как в разделах 7.4 и 7.6 использовался оператор области

видимости (`::`) для обращения к статическим членам и членам типа. В обычном коде (не шаблона) у компилятора есть доступ к определению класса. В результате он знает, является ли имя, к которому обращаются через оператор области видимости, типом или статическим членом. Например, в коде `string::size_type`, компилятор имеет определение класса `string` и может узнать, что `size_type` — это тип.

С учетом того, что `T` является параметром типа шаблона, когда компилятор встретит такой код, как `T::mem`, он не будет знать до времени создания экземпляра, является ли `mem` типом или статической переменной-членом. Но чтобы обработать шаблон, компилятор должен знать, представляет ли имя тип. Например, если `T` является именем параметра типа, то как компилятор воспримет следующий код:

```
T::size_type * p;
```

Он должен знать, определяется ли переменная по имени `p` или происходит умножение статической переменной-члена по имени `size_type` на переменную по имени `p`.

По умолчанию язык подразумевает, что имя, к которому обращаются через оператор области видимости, не является типом. В результате, если необходимо использовать тип-член параметра типа шаблона, следует явно указать компилятору, что имя является типом. Для этого используется ключевое слово `typename`:

```
template <typename T>
typename T::value_type top( const T& c ) {
    if ( !c.empty() )
        return c.back();
    else
        return typename T::value_type( );
}
```

Функция `top()` ожидает контейнер в качестве аргумента, она использует ключевое слово `typename` для определения своего типа возвращаемого значения и создает инициализированный по умолчанию элемент (см. раздел 7.5.3), чтобы возвратить его, если у контейнера с нет никаких элементов.



Когда необходимо уведомить компилятор о том, что имя представляет тип,

следует использовать ключевое слово `typename`, а не `class`.

Аргументы шаблона по умолчанию



Аналогично тому, как можно предоставить аргументы по умолчанию для параметров функции (см. раздел 6.5.1), можно предоставить *аргументы шаблона по умолчанию* (default template argument). По новому стандарту можно предоставлять аргументы по умолчанию и для шаблонов функций, и для шаблонов классов. Прежние версии языка допускали аргументы по умолчанию только для шаблонов класса.

В качестве примера перепишем функцию сравнения, использующую по умолчанию библиотечный шаблонный объект функции `less` (см. раздел 14.8.2):

```
// compare() имеет аргумент шаблона по умолчанию,
less<T>
// и заданный по умолчанию аргумент функции, F()
template <typename T, typename F = less<T>>
int compare(const T &v1, const T &v2, F f = F()) {
    if (f(v1, v2)) return -1;
    if (f(v2, v1)) return 1;
    return 0;
}
```

Здесь в шаблон добавлен второй параметр типа, `F`, представляющий тип вызываемого объекта (см. раздел 10.3.2), и определен новый параметр функции, `f`, который будет связан с вызываемым объектом.

Предоставлено также значение по умолчанию для этого параметра шаблона и соответствующего ему параметра функции. Аргумент шаблона по умолчанию определяет, что функция `compare()` будет использовать библиотечный класс `less` объекта функции, экземпляр которого создается с тем же параметром типа, что и функция `compare()`. Заданный по умолчанию аргумент функции указывает, что параметр `f` будет инициализирован по умолчанию объектом типа `F`.

Когда пользователи вызывают эту версию функции `compare()`, они могут предоставить собственный оператор сравнения, но не обязаны делать это:

```
bool i = compare(0, 42); // использует less; i
```

```
равно -1
// результат зависит от isbn в item1 и item2
Sales_data item1(cin), item2(cin);
bool j = compare(item1, item2, compareIsbn);
```

Первый вызов использует заданный по умолчанию аргумент функции, которым является объект типа `less<T>`. В этом вызове `T` имеет тип `int`, поэтому у объекта будет тип `less<int>`. Этот экземпляр функции `compare()` будет использовать для сравнения тип `less<int>`.

Во втором вызове передается функция `compareIsbn()` (см. раздел 11.2.2) и два объекта типа `Sales_data`. Когда функция `compare()` вызывается с тремя аргументами, типом третьего аргумента должен быть вызываемый объект, возвращающий тип, приводимый к типу `bool` и получающий аргументы типа, совместимого с типами первых двух аргументов. Как обычно, типы параметров шаблона выводятся из соответствующих им аргументов функции. В этом вызове тип `T` выводится как тип `Sales_data`, а тип `F` — как тип `compareIsbn()`.

Как и с аргументами функций по умолчанию, у параметра шаблона может быть аргумент по умолчанию, только если у всех параметров справа от него также есть аргументы по умолчанию.

Аргументы по умолчанию шаблона и шаблоны класса

Всякий раз, когда используется шаблон класса, за именем шаблона всегда должны следовать угловые скобки. Скобки означают, что класс будет создан как экземпляр шаблона. В частности, если шаблон класса предоставляет аргументы по умолчанию для всех своих параметров и следует использовать именно их, то после имени шаблона следует поместить пустую пару угловых скобок:

```
template <class T = int> class Numbers { // по
умолчанию T — это int
public:
    Numbers(T v = 0): val(v) {} // различные операции
с числами
private:
    T val;
};
Numbers<long double> lots_of_precision;
Numbers<> average_precision; // пустые <> означают
тип по умолчанию
```

Здесь создаются два экземпляра шаблона `Numbers`: версия `average_precision` — экземпляр `Numbers` с заменой параметра `T` типом `int`; версия `lots_of_precision` — экземпляр `Numbers` с заменой параметра `T` типом `long double`.

Упражнения раздела 16.1.3

Упражнение 16.17. Каковы (если есть) различия между параметром типа, объявленным с ключевым словом `typename` и ключевым словом `class`? Когда должно использоваться ключевое слово `typename`?

Упражнение 16.18. Объясните каждое из следующих объявлений шаблона функции и укажите, допустимы ли они. Исправьте все найденные ошибки.

- (a) `template <typename T, U, typename V> void f1(T, U, V);`
- (b) `template <typename T> T f2(int &T);`
- (c) `inline template <typename T> T foo(T, unsigned int*);`
- (d) `template <typename T> f4(T, T);`
- (e) `typedef char CType;`
`template <typename CType> CType f5(CType a);`

Упражнение 16.19. Напишите функцию, получающую ссылку на контейнер и выводящую его элементы. Используйте переменную `size_type` и функцию-член `size()` контейнера для контроля цикла, вывода элементов.

Упражнение 16.20. Перепишите функцию из предыдущего упражнения так, чтобы использовать для контроля цикла итераторы, возвращаемые функциями `begin()` и `end()`.

16.1.4. Шаблоны-члены

У класса (обычного или шаблона класса) может быть функция-член, которая сама является шаблоном. Такие члены называются *шаблонами-членами* (member template). Шаблоны-члены не могут быть виртуальными.

Шаблоны-члены обычных (не шаблонных) классов

В качестве примера обычного класса, у которого есть шаблон-член, определим класс, подобный стандартному типу функции удаления (`deleter`), используемой указателем `unique_ptr` (см. раздел 12.1.5). Как и у стандартной функции удаления, у данного класса будет перегруженный

оператор вызова функции (см. раздел 14.8), который, получив указатель, выполняет для него оператор `delete`. В отличие от стандартной функции удаления, новый класс будет также выводить сообщения при каждом запуске. Поскольку создаваемую функцию удаления предстоит использовать с любым типом, сделаем оператор вызова шаблоном:

```
// класс объекта функции,зывающий оператор
delete для указателя
class DebugDelete {
public:
    DebugDelete( std::ostream &s = std::cerr): os(s) {
}
// подобно любым шаблонам функции, тип T выводится
компилятором
template <typename T> void operator()( T *p) const
{
    os << "deleting unique_ptr" << std::endl;
delete p; }
private:
    std::ostream &os;
} ;
```

Как и любой другой шаблон, шаблон-член начинается с собственного списка параметров шаблона. У каждого объекта класса `DebugDelete` есть переменная-член типа `ostream` для вывода и функция-член, которая сама является шаблоном. Этот класс можно использовать вместо оператора `delete`:

```
double* p = new double;
DebugDelete d; // объект, способный действовать как
оператор delete
d( p); // вызывает DebugDelete::operator()( double*),
удаляющий p
int* ip = new int;
// вызывает operator()( int*) для временного объекта
DebugDelete
DebugDelete( )( ip);
```

Поскольку вызов объекта `DebugDelete` удаляет переданный ему указатель, его можно также использовать как функцию удаления для указателя `unique_ptr`. Чтобы переопределить функцию удаления указателя `unique_ptr`, укажем тип функции удаления в скобках и предоставим объект типа функции удаления конструктору (см. раздел

12.1.5):

```
// удалить объект, на который указывает p
// создает экземпляр DebugDelete::operator()<int>
(int *)
unique_ptr<int,           DebugDelete>      p( new      int,
DebugDelete());
// удаляет объект, на который указывает sp
// создает экземпляр DebugDelete::operator()
<string>( string*)
unique_ptr<string,        DebugDelete>     sp( new    string,
DebugDelete());
```

Здесь указано, что у функции удаления `p` будет тип `DebugDelete` и что предоставлен безымянный объект этого типа в конструкторе `p()`.

Деструктор класса `unique_ptr` вызывает оператор вызова типа `DebugDelete`. Таким образом, при каждом вызове деструктора класса `unique_ptr` создается также экземпляр оператора вызова класса `DebugDelete`. Таким образом, определения выше создадут следующие экземпляры:

```
// примеры создания экземпляров шаблонов-членов
DebugDelete
void DebugDelete::operator()( int *p) const { delete
p; }
void DebugDelete::operator()( string *p) const {
delete p; }
```

Шаблоны-члены шаблонов класса

Шаблон-член можно также определить и для шаблона класса. В данном случае у и класса, и у его члена будут собственные, независимые параметры шаблона.

В качестве примера снабдим класс `Blob` конструктором, который получает два итератора, обозначающих диапазон копируемых элементов. Поскольку желательно обеспечить поддержку итераторов в различных видах последовательностей, сделаем этот конструктор шаблоном:

```
template <typename T> class Blob {
    template <typename It> Blob( It b, It e);
    // ...
};
```

У этого конструктора есть свой собственный параметр типа шаблона,

`It`, который он использует для типа двух параметров функции.

В отличие от обычных функций-членов шаблонов класса, шаблоны-члены являются шаблонами функций. При определении шаблона-члена вне тела шаблона класса следует предоставить список параметров шаблона для шаблона класса и для шаблона функции. Список параметров для шаблона класса располагается сначала, затем следует список параметров шаблона-члена:

```
template <typename T> // параметр типа для класса
template <typename It> // параметр типа для
конструктора
Blob<T>::Blob( It b, It e) :
    data( std::make_shared<std::vector<T>>( b, e)) { }
```

Здесь определяется член шаблона класса, у которого есть один параметр типа шаблона `T`. Сам член является шаблоном функции, имеющий параметр типа `It`.

Создание экземпляров и шаблоны-члены

Чтобы создать экземпляр шаблона-члена шаблона класса, следует предоставить аргументы для параметров шаблона и класса, и функции. Как обычно, аргументы для параметров шаблона класса определяются типом объекта, через который происходит вызов шаблона-члена. Так же как обычно, компилятор, как правило, выводит тип аргументов шаблона для собственных параметров шаблона-члена из аргументов, переданных при вызове (см. раздел 16.1.1):

```
int ia[] = { 0,1,2,3,4,5,6,7,8,9 };
vector<long> vi = { 0,1,2,3,4,5,6,7,8,9 };
list<const char*> w = { "now", "is", "the", "time" };
// создает экземпляр класса Blob<int>
// и конструктор Blob<int> с двумя параметрами типа
int*
Blob<int> a1( begin( ia), end( ia));
// создает экземпляр конструктора Blob<int> с двумя
параметрами
// типа vector<long>::iterator
Blob<int> a2( vi.begin(), vi.end());
// создает экземпляр класса Blob<string> и
конструктор Blob<string>
// с двумя параметрами типа list<const
char*>::iterator
```

```
Blob<string> a3( w.begin( ), w.end( ) );
```

При определении `a1` указывается явно, что компилятор должен создать экземпляр шаблона `Blob` с параметром типа `int`. Параметр типа для его собственных параметров конструктора будет выведен из типа результатов вызова функций `begin(ia)` и `end(ia)`. Этим типом является `int*`. Таким образом, определение `a1` создает следующий экземпляр:

```
Blob<int>::Blob( int*, int* );
```

Определение `a2` использует уже готовый экземпляр класса `Blob<int>` и создает экземпляр конструктора с параметром типа `It`, замененным на `vector<short>::iterator`. Определение `a3` (явно) создает экземпляр шаблона `Blob` с собственным параметром шаблона типа `string` и (неявно) экземпляр конструктора шаблона-члена этого класса с собственным параметром типа `list<const char*>`.

Упражнения раздела 16.1.4

Упражнение 16.21. Напишите собственную версию типа `DebugDelete`.

Упражнение 16.22. Пересмотрите программы `TextQuery` из раздела 12.3 так, чтобы указатель-член `shared_ptr` использовал тип `DebugDelete` как свою функцию удаления (см. раздел 12.1.4).

Упражнение 16.23. Предскажите, когда будет выполняться оператор вызова в вашей основной программе запроса. Если предсказание неправильно, убедитесь, что понимаете почему.

Упражнение 16.24. Добавьте в свой шаблон `Blob` конструктор, получающий два итератора.



16.1.5. Контроль создания экземпляра

Тот факт, что экземпляр шаблона создается только при его использовании (см. раздел 16.1.1), означает, что создание того же экземпляра может происходить в нескольких объектных файлах. Когда два или более отдельно откомпилированных файла исходного кода используют тот же шаблон с теми же аргументами шаблона, создание экземпляра этого шаблона осуществляется в каждом из этих файлов.



В больших системах дополнительные затраты на создание экземпляра того

же шаблона в нескольких файлах могут оказаться существенными. По новому стандарту можно избежать этих дополнительных затрат за счет явного создания экземпляра (*explicit instantiation*). Его форма такова:

```
extern template объявление; // объявление создания
экземпляра
template объявление;           // определение создания
экземпляра
```

где *объявление* — это объявление класса или функции, в котором все параметры шаблона заменены аргументами шаблона. Например:

```
// объявление и определение создания экземпляра
extern template class Blob<string>;                      //
объявление
template int compare(const int&, const int&); // //
определение
```

Когда компилятор встретит внешнее (*extern*) объявление шаблона, он не будет создавать код его экземпляра в этом файле. Объявление экземпляра как *extern* является обещанием того, что будет и не внешнее создание экземпляра в другом месте программы. Вполне может быть несколько внешних объявлений для каждого экземпляра, однако по крайней мере одно определение экземпляра должно быть.

Поскольку компилятор автоматически создает экземпляр шаблона при его использовании, объявление *extern* должно располагаться перед любым кодом, который использует этот экземпляр:

```
// Application.cc
// экземпляры этих шаблонов должны быть созданы
// в другом месте программы
extern template class Blob<string>;
extern template int compare(const int&, const
int&);
Blob<string> sal, sa2; // экземпляр создается в
другом месте
// экземпляры Blob<int> и его конструктор
initializer_list создаются
// в этом файле
Blob<int> a1 = {0,1,2,3,4,5,6,7,8,9};
Blob<int> a2(a1); // экземпляр конструктора копий
// создается в этом файле
int i = compare(a1[0], a2[0]); // экземпляр
```

создается в другом месте

Файл Application.о будет создавать экземпляр класса Blob<int> наряду с его конструктором initializer_list и конструктором копий. Экземпляры функции compare<int> и класса Blob<string> не будут созданы в этом файле. Определения этих шаблонов должны быть в каком-то другом файле программы:

```
// templateBuild.cc  
// файл создания экземпляра должен предоставить  
обычное определение для  
// каждого типа и функции, которые другие файлы  
объявляют внешними  
template int compare( const int&, const int& );  
template class Blob<string>; // создает экземпляры  
всех членов  
// шаблона класса
```

В отличие от объявления, когда компилятор видит определение экземпляра, он создает код. Таким образом, файл templateBuild.о будет содержать определения функции compare() для экземпляра типа int и класса Blob<string>. При построении приложения следует скомпоновать файл templateBuild.о с файлом Application.о.



Для каждого объявления экземпляра где-нибудь в программе должно быть определение явного создания экземпляра.

Определения экземпляров создают экземпляры всех членов

Определение экземпляра для шаблона класса создает экземпляры всех членов этого шаблона, включая встраиваемые функции-члены. Когда компилятор видит определение экземпляра, он не может знать, какие функции-члены использует программа. Следовательно, в отличие от обычного способа создания экземпляра шаблона класса, компилятор создает экземпляры всех членов этого класса. Даже если член класса не будет использоваться, его экземпляр будет создан все равно. Следовательно, явное создание экземпляра можно использовать только для таких типов, которые применимы со всеми членами данного шаблона.



Определение экземпляра используется только для таких типов, которые применимы со всеми функциями-членами шаблона класса.

Упражнения раздела 16.1.5

Упражнение 16.25. Объясните значение этих объявлений:

```
extern template class vector<string>;
template class vector<Sales_data>;
```

Упражнение 16.26. Предположим, что класс `NoDefault` не имеет стандартного конструктора. Можно ли явно создать экземпляр `vector<NoDefault>`? Если нет, то почему?

Упражнение 16.27. Объясните по каждому помеченному оператору, происходит ли создание экземпляра. Если создается экземпляр шаблона, объясните, почему; если нет, то тоже почему.

```
template <typename T> 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)
}
```



16.1.6. Эффективность и гибкость

Библиотечные типы интеллектуальных указателей (см. раздел 12.1) являются хорошим примером грамотно спроектированных шаблонов.

Очевидное различие между указателями `shared_ptr` и `unique_ptr` в стратегии, которую они используют для управления содержащимися в них указателями: один класс предоставляет совместную собственность; а

другой — единоличною собственность на хранимый указатель. Это различие и является основанием для создания данных классов.

Данные классы отличаются также тем, как они позволяют пользователям переопределять свою стандартную функцию удаления. Для переопределения функции удаления класса `shared_ptr` достаточно предоставить ему при создании вызываемый объект или функцию `reset()`. У объекта класса `unique_ptr`, напротив, тип функции удаления является частью типа. При определении указателя `unique_ptr` пользователи должны предоставлять этот тип как явный аргумент шаблона. В результате для указателя `unique_ptr` сложней предоставить собственную функцию удаления.

Различие в способе работы функции удаления — это лишь частность функциональных возможностей данных классов. Но, как будет вскоре продемонстрировано, это различие в стратегии реализации может серьезно повлиять на производительность.

Привязка функции удаления во время выполнения

Даже не зная, как именно реализуются библиотечные типы, вполне можно догадаться, что указатель `shared_ptr` обращается к своей функции удаления косвенно. Поэтому функция удаления должна храниться как указатель или как класс (такой как `function` из раздела 14.8.3), инкапсулирующий указатель.

То, что тип функции удаления не известен до времени выполнения, позволяет убедиться, что класс `shared_ptr` не содержит функцию удаления как непосредственный член класса. Действительно, класс `shared_ptr` позволяет изменить тип функции удаления на протяжении продолжительности его существования. Вполне можно создать указатель `shared_ptr`, используя функцию удаления одного типа, а впоследствии использовать функцию `reset()`, чтобы использовать для того же указателя `shared_ptr` другой тип функции удаления. Вообще, у класса не может быть члена, тип которого изменяется во время выполнения. Следовательно, функция удаления должна храниться отдельно.

Размышляя о том, как должна работать функция удаления, предположим, что класс `shared_ptr` хранит контролируемый указатель в переменной-члене класса по имени `p`, а обращение к функции удаления осуществляется через член класса по имени `del`. Деструктор класса `shared_ptr` должен включать такой оператор:

```
// значение del станет известно только во время
```

```
выполнения; вызов
// через указатель
del ? del(p) : delete p; // вызов del (p) требует
перехода во время
// выполнения к области
хранения del
```

Поскольку функция удаления хранится отдельно, вызов `del(p)` требует перехода во время выполнения к области хранения `del` и выполнения кода, на который он указывает.

Привязка функции удаления во время компиляции

Теперь давайте подумаем, как мог бы работать класс `unique_ptr`. В этом классе тип функции удаления является частью типа `unique_ptr`. Таким образом, у шаблона `unique_ptr` есть два параметра шаблона: представляющий контролируемый указатель и представляющий тип функции удаления. Поскольку тип функции удаления является частью типа `unique_ptr`, тип функции-члена удаления известен на момент компиляции. Функция удаления может храниться непосредственно в каждом объекте класса `unique_ptr`.

Деструктор класса `unique_ptr` работает подобно таковому у класса `shared_ptr`, в котором он вызывает предоставленную пользователем функцию удаления или выполняет оператор `delete` для хранимого указателя:

```
// del связывается во время компиляции; создается
экземпляр прямого
// вызова функции удаления
del(p); // нет дополнительных затрат во время
выполнения
```

Тип `del` — это либо заданный по умолчанию тип функции удаления, либо тип, предоставленный пользователем. Это не имеет значения; так или иначе, выполняемый код будет известен во время компиляции. Действительно, если функция удаления похожа на класс `DebugDelete` (см. раздел 16.1.4), этот вызов мог бы даже быть встраиваемым во время компиляции.

При привязке функции удаления во время компиляции класс `unique_ptr` избегает во время выполнения дополнительных затрат на косвенный вызов своей функции удаления. При привязке функции удаления во время выполнения класс `shared_ptr` облегчает

пользователю переопределение функции удаления.

Упражнения раздела 16.1.6

Упражнение 16.28. Напишите собственные версии классов `shared_ptr` и `unique_ptr`.

Упражнение 16.29. Пересмотрите свой класс `Blob` так, чтобы использовать собственную версию класса `shared_ptr`, а не библиотечную.

Упражнение 16.30. Повторно выполните некоторые из своих предыдущих программ, чтобы проверить собственные переделанные классы `shared_ptr` и `Blob`. (Примечание: реализация типа `weak_ptr` не рассматривается в этом издании, поэтому не получится использовать класс `BlobPtr` с пересмотренным классом `Blob`.)

Упражнение 16.31. Объясните, как компилятор мог бы встроить вызов функции удаления, если бы с классом `unique_ptr` был использован класс `DebugDelete`.

16.2. Дедукция аргумента шаблона

Как уже упоминалось, для определения параметров шаблона для шаблона функции компилятор по умолчанию использует аргументы в вызове. Процесс определения аргументов шаблона по аргументам функции называется дедукцией аргумента шаблона (template argument deduction). В ходе дедукции аргумента шаблона компилятор использует типы аргументов вызова для поиска таких аргументов шаблона, которые обеспечат лучшее соответствие создаваемой версии функции для данного вызова.



16.2.1. Преобразования и параметры типа шаблона

Подобно нешаблонным функциям, передаваемые в вызове шаблона функции аргументы используются для инициализации параметров этой функции. Параметры функции, тип которых использует параметр типа шаблона, имеют специальные правила инициализации. Только очень ограниченное количество автоматических преобразований применимо к таким аргументам. Вместо преобразования аргументов компилятор создает новые экземпляры.

Как обычно, спецификаторы `const` верхнего уровня (см. раздел 2.4.3) в параметре или аргументе игнорируются. Единственными остальными преобразованиями, выполняемыми при вызове шаблона функции, являются следующие.

- Преобразования констант: параметр функции, являющийся ссылкой (или указателем) на константу, может быть передан как ссылка (или указатель) на не константный объект (см. раздел 4.11.2).
- Преобразование массива или функции в указатель: если тип параметра функции не будет ссылочным, то к аргументам типа массива или функции будет применено обычное преобразование указателя. Аргумент типа массива будет преобразован в указатель на его первый элемент. Точно так же аргумент типа функции будет преобразован в указатель на тип функции (см. раздел 4.11.2).

Другие преобразования, такие как арифметические преобразования (см. раздел 4.11.1), преобразования производного в базовый (см. раздел 15.2.2) и пользовательские преобразования (см. разделы 7.5.4 и 14.9) не

выполняются.

В качестве примера рассмотрим вызовы функции `fobj()` и `fre()`. Функция `fobj()` копирует свои параметры, тогда как параметры функции `fre()` являются ссылками:

```
template <typename T> T fobj( T, T); // аргументы  
копируются  
template <typename T> T fre( const T&, const T&);  
// ссылки  
string s1("a value");  
const string s2("another value");  
fobj(s1, s2); // вызов fobj(string, string); const  
игнорируется  
fre(s1, s2); // вызов fre(const strings, const  
string&) использует  
// допустимое преобразования в  
константу для s1  
int a[10], b[42];  
fobj(a, b); // вызов f(int*, int*)  
fre(a, b); // ошибка: типы массивов не совпадают
```

В первой паре вызовов как аргументы передаются строка и константная строка. Даже при том, что эти типы не соответствуют точно друг другу, оба вызова допустимы. В вызове функции `fobj()` аргументы копируются, поэтому не имеет значения, был ли первоначальный объект константой. В вызове функции `fre()` тип параметра — ссылка на константу. Преобразование в константу для ссылочного параметра является разрешенным преобразованием, поэтому данный вызов допустим.

В следующей паре вызовов как аргументы передаются массивы, отличающиеся размером, а следовательно, имеющие разные типы. В вызове функции `fobj()` различие типов массивов не имеет значения. Оба массива преобразуются в указатели. Типом параметра шаблона в функции `fobj` является `int*`. Вызов функции `fre()`, однако, недопустим. Когда параметр является ссылкой, массивы не преобразовываются в указатели (см. раздел 6.2.4). Типы `a` и `b` не совпадают, поэтому вызов ошибочен.



Единственными допустимыми автоматическими преобразованиями для

аргументов в параметры типа шаблонов являются преобразования константы в массив или функций в указатель.

Параметры функций с одинаковым типом параметра шаблона

Параметр типа шаблона применим как тип нескольких параметров функции. Поскольку набор преобразований ограничен, аргументы таких параметров должны быть, по существу, того же типа. Если выведенные типы не совпадают, то вызов ошибочен. Например, функция `compare()` (см. раздел 16.1.1) получает два параметра `const T&`. У ее аргументов должен быть фактически тот же тип:

```
long lng;  
compare(lng, 1024); // ошибка: нельзя создать  
                     // экземпляр compare(long, int)
```

Этот вызов ошибочен потому, что у аргументов функции `compare()` не совпадают типы. Для первого аргумента выведен аргумент шаблона типа `long`; а для второго — `int`. Эти типы не совпадают, поэтому дедукция аргумента шаблона терпит неудачу.

Если необходимо обеспечить обычные преобразования аргументов, можно определить функцию с двумя параметрами типа:

```
// типы аргумента могут отличаться, но должны быть  
совместимы
```

```
template <typename A, typename B>  
int flexibleCompare(const A& v1, const B& v2) {  
    if (v1 < v2) return -1;  
    if (v2 < v1) return 1;  
    return 0;  
}
```

Теперь пользователь может предоставлять аргументы разных типов:

```
long lng;  
flexibleCompare(lng, 1024); // ok: вызов  
flexibleCompare(long, int)
```

Конечно, должен существовать оператор `<`, способный сравнивать значения этих типов.

Обычные преобразования применимы к обычным аргументам

У шаблона функции могут быть параметры, определенные с использованием обычных типов, т.е. типов, которые не задействуют

параметр типа шаблона. Такие аргументы не обрабатываются специальным образом; они преобразуются, как обычно, в соответствующий тип параметра (см. раздел 6.1). Рассмотрим, например, следующий шаблон:

```
template <typename T> ostream &print(ostream &os,
const T &obj) {
    return os << obj;
}
```

Тип первого параметра функции известен: `ostream&`. У второго параметра, `obj`, тип параметра шаблона. Поскольку тип параметра `os` фиксирован, при вызове функции `print()` к переданным ему аргументам применимы обычные преобразования:

```
print(cout,      42);           // создает экземпляр
print(ostream&, int)
ofstream f("output");
print(f, 10); // использует print(ostream&, int);
              // преобразует f в ostream&
```

В первом вызове тип первого аргумента точно соответствует типу первого параметра. Этот вызов задействует ту версию функции `print()`, которая получает тип `ostream&` и тип `int` для создания экземпляра. Во втором вызове первый аргумент имеет тип `ofstream`, а преобразование из `ofstream` в `ostream&` допустимо (см. раздел 8.2.1). Поскольку тип этого параметра не зависит от параметра шаблона, компилятор неявно преобразует `f` в `ostream&`.



Обычные преобразования применимы к аргументам, тип которых не является параметром шаблона.

Упражнения раздела 16.2.1

Упражнение 16.32. Что происходит при дедукции аргумента шаблона?

Упражнение 16.33. Назовите два преобразования типов, допустимых для аргументов функций, при дедукции аргумента шаблона.

Упражнение 16.34. С учетом только следующего кода объясните, допустим ли каждый из этих вызовов. Если да, то каков тип `T`? Если нет, то почему?

```
template <class T> int compare(const T&, const T&);
```

```
( a) compare( "hi",    "world");   ( b) compare( "bye",  
"dad");
```

Упражнение 16.35. Какой из следующих вызовов ошибочен (если он есть)? Каков тип Т допустимых вызовов? В чем проблема недопустимых вызовов?

```
template <typename T> T calc( T, int);  
template <typename T> T fcn( T, T);  
double d; float f; char c;  
( a) calc(c, 'c'); ( b) calc(d, f);  
( c) fcn(c, 'c'); ( d) fcn(d, f);
```

Упражнение 16.36. Что происходит при следующих вызовах:

```
template <typename T> f1( T, T);  
template <typename T1, typename T2> f2( T1, T2);  
int i = 0, j = 42, *p1 = &i, *p2 = &j;  
const int *cp1 = &i, *cp2 = &j;  
( a) f1( p1, p2); ( b) f2( p1, p2); ( c) f1( cp1,  
cp2);  
( d) f2( cp1, cp2); ( e) f1( p1, cp1); ( e) f2( p1, cp1);
```



16.2.2. Явные аргументы шаблона функции

В некоторых редких случаях компилятор не может вывести типы аргументов шаблона. В других случаях следует позволить пользователю контролировать создание экземпляра шаблона. Оба эти случая наиболее вероятны тогда, когда тип возвращаемого значения функции отличается от типов используемых ею параметров.

Определение явного аргумента шаблона

В качестве примера случая, когда необходимо позволить пользователю задавать тип, определим шаблон функции `sum()`, получающий аргументы двух разных типов. Тип результата будет определять пользователь. Таким образом, пользователь сможет выбрать необходимую ему точность.

Чтобы предоставить пользователю контроль над типом возвращаемого значения, определим третий параметр шаблона, представляющий тип возвращаемого значения:

```
// тип T1 не может быть выведен: он отсутствует в
```

списке параметров

```
// функции
template <typename T1, typename T2, typename T3>
T1 sum( T2, T3);
```

В данном случае нет никакого аргумента, тип которого мог бы использоваться для выводения типа T1. Для этого параметра при каждом вызове функции sum()зывающая сторона должна предоставить *явный аргумент шаблона* (explicit template argument).

Явный аргумент шаблона предоставляется вызову тем же способом, что и экземпляру шаблона класса. Явные аргументы шаблона определяются в угловых скобках после имени функции и перед списком аргументов:

```
// T1 определяется явно; T2 и T3 выводятся из типов
аргумента
```

```
auto val3 = sum<long long>( i, lng); // long long
sum( int, long)
```

Этот вызов явно определяет тип параметра T1. Компилятор выведет типы для параметров T2 и T3 из типов переменных i и lng.

Явные аргументы шаблона отвечают соответствующим параметрам шаблона слева направо; первый аргумент шаблона отвечает первому параметру шаблона, второй аргумент — второму параметру и т.д. Явный аргумент шаблона может быть пропущен только для замыкающих (крайних справа) параметров, и то, только если они могут быть выведены из параметров функции. Если функция sum() была написана следующим образом:

```
// плохой проект: пользователи вынуждены явно
определять все три
// параметра шаблона
template <typename T1, typename T2, typename T3>
T3 alternative_sum( T2, T1);
```

то пользователям придется всегда определять аргументы для всех трех параметров:

```
// ошибка: нельзя вывести начальные параметры
шаблона
auto val3 = alternative_sum<long long>( i, lng);
// ok: все три параметра определяются явно
auto val2 = alternative_sum<long long, int, long>
( i, lng);
```

Нормальные преобразования применимы к аргументам,

определенным явно

По тем же причинам, по которым нормальные преобразования разрешены для параметров, определенных с использованием обычных типов (см. раздел 16.2.1), нормальные преобразования применимы также для аргументов, параметры типа шаблона которых определяются явно:

```
long lng;  
compare(lng, 1024); // ошибка: параметры  
шаблона не совпадают  
compare<long>(lng, 1024); // ok: создает экземпляр  
compare( long, long)  
compare<int>(lng, 1024); // ok: создает экземпляр  
compare( int, int)
```

Как уже упоминалось, первый вызов ошибочен, поскольку у аргументов функции `compare()` должен быть одинаковый тип. Если тип параметра шаблона определен явно, обычные преобразования вполне применимы. Таким образом, вызов `compare<long>()` эквивалентен вызову функции, получающей два параметра `const long&`. Параметр типа `int` автоматически преобразуется в тип `long`. Во втором вызове параметр `T` явно определяется как тип `int`, таким образом, тип аргумента `lng` преобразовывается в `int`.

Упражнения раздела 16.2.2

Упражнение 16.37. Библиотечная функция `max()` имеет два параметра функции и возвращает больший из своих аргументов. У этой функции есть один параметр типа шаблона. Можно ли вызвать функцию `max()`, передав ей аргументы типа `int` и `double`? Если да, то как? Если нет, то почему?

Упражнение 16.38. Когда происходит вызов функции `make_shared()` (см. раздел 12.1.1), следует предоставить явный аргумент шаблона. Объясните, почему этот аргумент необходим и как он используется.

Упражнение 16.39. Используйте явный аргумент шаблона, чтобы сделать возможной передачу двух строковых литералов первоначальной версии функции `compare()` из раздела 16.1.1.



16.2.3. Замыкающие типы возвращаемого значения и трансформация типа

Применение явного аргумента шаблона для представления типа возвращаемого значения шаблона функции хорошо работает тогда, когда необходимо позволить пользователю определять тип возвращаемого значения. В других случаях обязательное предоставление явного аргумента шаблона налагает дополнительное бремя на пользователя без всяких преимуществ. Например, можно написать функцию, которая получает два обозначающих последовательность итератора и возвращает ссылку на элемент этой последовательности:

```
template <typename It>
??? & fcn( It beg, It end) {
    // обработка диапазона
    return *beg; // возвратить ссылку на элемент из
диапазона
}
```

Точный тип, подлежащий возвращению, неизвестен, но известно, что он будет ссылкой на тип элемента обрабатываемой последовательности:

```
vector<int> vi = {1,2,3,4,5};
Blob<string> ca = { "hi", "bye" };
auto &i = fcn( vi.begin(), vi.end() ); // fcn()
должна возвратить int&
auto &s = fcn( ca.begin(), ca.end() ); // fcn()
должна возвратить string&
```



Здесь известно, что функция возвратит ссылку `*beg`, а также, что можно использовать выражение `decltype(*beg)` для получения типа этого выражения. Однако параметр `beg` не существует, пока не встретится список параметров. Чтобы определить эту функцию, следует использовать замыкающий тип возвращаемого значения (см. раздел 6.3.3). Поскольку замыкающий тип располагается после списка параметров, он может использовать параметры функции:

```
// замыкающий тип позволяет объявлять тип
возвращаемого значения уже
// после списка параметров
template <typename It>
auto fcn( It beg, It end) -> decltype(*beg) {
    // обработка диапазона
    return *beg; // возвратить ссылку на элемент из
```

диапазона

}

Здесь компилятору было указано, что тип возвращаемого значения функции `fcn()` совпадает с типом, возвращенным при обращении к значению параметра `beg`. Оператор обращения к значению возвращает 1-значение (см. раздел 4.1.1), таким образом, выведенный выражением `decltype` тип является ссылкой на тип элемента, обозначаемого параметром `beg`. Следовательно, если функция `fcn()` будет вызвана для последовательности строк, то типом возвращаемого значения будет `string&`. Если это будет последовательность элементов типа `int`, то возвращен будет тип `int&`.

Трансформация типа классов библиотечных шаблонов

Иногда прямого доступа к необходимому типу нет. Например, может возникнуть необходимость в функции, подобной `fcn()`, которая возвращает элемент по значению (см. раздел 6.3.2), а не по ссылке.

Проблема написания этой функции в том, что о передаваемых типах неизвестно почти ничего. Единственные известные в этой функции операции, которые можно использовать, — это операции с итераторами, и нет никаких операций с итераторами, которые возвращают элементы (в противоположность ссылкам на элементы).

Чтобы получить тип элемента, можно использовать библиотечный шаблон *трансформации типа* (type transformation). Эти шаблоны определяются в заголовке `type_traits`. Обычно классы заголовка `type_traits` используются для так называемого шаблонного метaprogramмирования, не рассматриваемого в данной книге. Однако шаблоны трансформации типа полезны и в обычном программировании. Они описаны в табл. 16.1, а их реализация рассматривается в разделе 16.5 (стр. 892).

В данном случае для получения типа элемента можно использовать шаблон `remove_reference`. У шаблона `remove_reference` один параметр типа шаблона и (открытый) тип-член `type`. Если экземпляр шаблона `remove_reference` создается со ссылочным типом, то тип `type` будет ссылочным. Например, если создать экземпляр `remove_reference<int&>`, то типом `type` будет `int`. Точно так же, если создать экземпляр `remove_reference<string&>`, то типом `type` будет `string` и т.д. Таким образом, при условии, что `beg` — итератор, следующее выражение возвратит тип элемента, на который указывает

итератор beg:

```
remove_reference<decltype( *beg) >::type
```

Выражение decltype(*beg) возвратит ссылочный тип элемента type. Выражение remove_reference::type удаляет ссылку, оставляя тип самого элемента.

Таблица 16.1. Стандартные шаблоны трансформации типа

Для Mod<T>, где Mod есть	Если T есть	To Mod<T>::type есть
remove_reference	X& ИЛИ X&&	X
	в противном случае	T
add_const	X&, const X или функция	T
	в противном случае	const T
add_l-value_reference	X&	T
	X&&	X&
	в противном случае	T&
add_r-value reference	X& ИЛИ X&&	T
	в противном случае	T&&
remove_pointer	X*	X
	в противном случае	T
add_pointer	X& ИЛИ X&&	X*
	в противном случае	T*
make_signed	unsigned X	X
	в противном случае	T
make_unsigned	знаковый тип	unsigned T
	в противном случае	T
remove_extent	X[n]	X
	в противном случае	T
remove_all_extents	X[n1][n2]...	X
	в противном случае	T

Используя шаблон remove_reference и замыкающий тип с выражением decltype, можно написать собственную функцию, возвращающую копию значения элемента:

```
// для использования типа-члена параметра шаблона  
следует
```

```
// использовать type_name; см. р. 16.1.3
```

```

template <typename It> auto fcn2(It beg, It end) ->
    typename remove_reference<decltype(*beg)>::type {
    // обработка диапазона
    return *beg; // возвратить копию элемента из
диапазона
}

```

Обратите внимание, что тип-член `type` зависит от параметра шаблона. Таким образом, чтобы указать компилятору, что `type` представляет тип (см. раздел 16.1.3), в объявлении типа возвращаемого значения следует использовать ключевое слово `typename`.

Каждый из описанных в табл. 16.1 шаблонов трансформации типа работает так же, как шаблон `remove_reference`. У каждого шаблона есть открытый член `type`, представляющий тип. Этот тип может быть связан с собственным параметром типа шаблона способом, о котором свидетельствует имя шаблона. Если невозможно (или ненужно) преобразовать параметр шаблона, тип-член `type` имеет тип параметра самого шаблона. Например, если `T` — это тип указателя, то `remove_pointer<T>::type` возвращает тип, на который указывает указатель `T`. Если `T` не указатель, то никакого преобразования не нужно. В данном случае у типа `type` тот же тип, что и у `T`.

Упражнения раздела 16.2.3

Упражнение 16.40. Корректна ли следующая функция? Если нет, то почему? Если она допустима, то каковы ограничения на типы ее аргументов (если они есть) и каков тип возвращаемого значения?

```

template <typename It>
auto fcn3(It beg, It end) -> decltype(*beg + 0) {
    // обработка диапазона
    return *beg; // возвратить копию элемента из
диапазона
}

```

Упражнение 16.41. Напишите версию функции `sum()` с типом возвращаемого значения, который будет гарантированно большим, чтобы содержать результат сложения.



16.2.4. Указатели на функцию и дедукция аргумента

При инициализации или присвоении указателя на функцию (см. раздел 6.7) из шаблона функции для вывода аргументов шаблона компилятор использует тип указателя.

Предположим, например, что есть указатель на функцию, которая возвращает тип `int` и получает два параметра, каждый из которых является ссылкой на `const int`. Этот указатель можно использовать для указания на экземпляр функции `compare()`:

```
template <typename T> int compare( const T&, const T& );
// pf1 указывает на экземпляр int compare( const int&, const int& )
int (*pf1)( const int&, const int&) = compare;
```

Тип параметров `pf1` определяет тип аргумента шаблона для параметра `T`. Аргументом шаблона для параметра `T` будет `int`. Указатель `pf1` указывает на экземпляр функции `compare()` с параметром `T`, связанным с типом `int`. Если аргументы шаблона не могут быть выведены из типа указателя функции, произойдет ошибка:

```
// перегруженные версии func(); каждая получает
разный тип указателя
// функции
void func( int(*)( const string&, const string&));
void func( int(*)( const int&, const int&));
func( compare); // ошибка: какой из экземпляров
compare?
```

Проблема в том, что, глядя на тип параметра функции `func()`, невозможно определить уникальный тип для аргумента шаблона. Вызов функции `func()` мог бы создать экземпляр версии функции `compare()`, получающей целые числа или версию, получающую строки. Поскольку невозможно идентифицировать уникальный экземпляр для аргумента функции `func()`, этот вызов не будет откомпилирован.

Неоднозначность вызова функции `func()` можно устраниТЬ при помощи явных аргументов шаблона:

```
// ok: явно определенная версия экземпляра
compare()
func( compare<int>); // передача compare( const int&, const int&)
```

Это выражение вызывает версию функции `func()`, получающую указатель на функцию с двумя параметрами типа `const int&`.



Когда возвращается адрес экземпляра шаблона функции, контекст должен позволять однозначно идентифицировать тип или значение для каждого параметра шаблона.



16.2.5. Дедукция аргумента шаблона и ссылки

Чтобы лучше понять дедукцию типа, рассмотрим такой вызов функции где параметр функции `p` является ссылкой на параметр типа шаблона `T`:

```
template <typename T> void f( T &p );
```

Обратите внимание на два момента: здесь применяются обычные правила привязки ссылок; и спецификаторы `const` здесь нижнего уровня, а не верхнего.

Дедукция типа из параметров ссылки на l-значения функций

Когда параметр функции представляет собой обычную ссылку (`l`-значение) на параметр типа шаблона (т.е. имеющего форму `T&`), правила привязки гласят, что передавать можно только `l`-значения (например, переменная или выражение, возвращающее ссылочный тип). Этот аргумент может быть или не быть константным. Если аргумент будет константой, то тип `T` будет выведен как константный:

```
template <typename T> void f1( T& ); // аргумент  
должен быть l-значением  
// вызовы f1() используют ссылочный тип аргумента  
как тип параметра  
// шаблона  
f1( i ); // i - это int; параметр шаблона T - это  
int  
f1( ci ); // ci - это const int; параметр шаблона T -  
это const int  
f1( 5 ); // ошибка: аргумент ссылочного параметра  
// должен быть l-значением
```

Если параметр функции имеет тип `const T&`, обычные правила привязки гласят, что можно передать любой вид аргумента — объект (константный или нет), временный объект или литеральное значение. Когда сам параметр функции является константой, выведенный для параметра `T` тип не будет константным типом. Константность является частью типа параметра функции, и поэтому она не становится также частью типа параметра шаблона:

```
template <typename T> void f2( const T& ); // может  
получать r-значения
```

```

// параметр в f2() - это const &; const в аргументе
неуместен
// в каждом из этих трех вызовов параметр функции
f2() выводится
// как const int&
f2( i ); // i - это int; параметр шаблона T - это
int
f2( ci ); // ci - это const int, но параметр шаблона
T - это int
f2( 5 ); // параметр const & может быть привязан к
r-значению;
// T - это int

```

Дедукция типа из параметров ссылки на r-значения функций

Когда параметр функции является ссылкой на r-значение (см. раздел 13.6.1), т.е. имеет форму $T\&\&$, обычные правила привязки гласят, что этому параметру можно передать r-значение. При этом дедукция типа ведет себя таким же образом, как дедукция обычного ссылочного параметра функции на l-значение. Выведенный тип для параметра T — это тип r-значения:

```

template <typename T> void f3( T&& );
f3( 42 ); // аргумент - это r-значение типа int;
параметр
// шаблона T - это int

```

Сворачивание ссылок и параметры ссылок на r-значения

Предположим, что i является объектом типа int . Можно подумать, что такой вызов, как $f3(i)$, будет недопустим. В конце концов, i — это l-значение, а ссылку на r-значение обычно нельзя связать с l-значением. Однако язык определяет два исключения из обычных правил привязки, которые позволяют это. На этих исключениях из правил основан принцип работы таких библиотечных функций, как $move()$.

Первое исключение относится к дедукции типа для ссылочного параметра на r-значение. Когда l-значение (например, i) передается параметру функции, являющемуся ссылкой на r-значение на параметр типа шаблона (например, $T\&\&$), компилятор выводит параметр типа шаблона как тип ссылки на l-значение аргумента. Поэтому, когда происходит вызов $f3(i)$, компилятор выводит тип T как $int\&$, а не int .

Выведение типа T как $int\&$, казалось бы, означает, что параметр

функции `f3()` будет ссылкой на `l`-значение типа `int&`. Обычно нельзя (непосредственно) определить ссылку на ссылку (см. раздел 2.3.1). Но это можно сделать косвенно, через псевдоним типа (см. раздел 2.5.1) или через параметр типа шаблона.



В таких ситуациях проявляется второе исключение из обычных правил привязки: если косвенно создать ссылку на ссылку, то эти ссылки "сворачиваются" (collapse). Во всех случаях кроме одного сворачивание ссылок формирует обычный тип ссылки на `l`-значение. Новый стандарт дополняет правила свертывания, включая ссылки на `l`-значение. Ссылки сворачиваются, формируя ссылку на `l`-значение только в специфическом случае ссылки на `l`-значение на ссылку на `l`-значение. Таким образом, для данного типа `X`:

- `X& &, X& && и X&& &` сворачиваются в тип `X&`.
- Тип `X&& &&` сворачивается в тип `X&&`.



Сворачивание ссылок применимо только тогда, когда ссылка на ссылку создается косвенно, как в псевдониме типа или параметре шаблона.

Комбинация правил свертывания ссылок и специального правила дедукции типа для ссылочных на `l`-значения параметров означает, что можно вызвать функцию `f3()` для `l`-значения. Когда параметру функции `f3()` (ссылке на `l`-значение) передается `l`-значение, компилятор выведет тип `T` как тип ссылки на `l`-значение:

```
f3(i); // аргумент - это l-значение; параметр T
шаблона - это int&
f3(ci); // аргумент - это l-значение;
// параметр T шаблона - это const int&
```

Когда параметр `T` шаблона выводится как ссылочный тип, правило свертывания гласит, что параметр функции `T&&` сворачивается в тип ссылки на `l`-значение. Например, результирующий экземпляр для вызова `f3(i)` получится примерно таким:

```
// недопустимый код, приведен только для примера
void f3<int&>(int& &&); // когда T - это int&,
```

параметр

// функции — это `int& &&`

Параметр функции `f3()` — это `T&&`, а `T` — это `int&`, таким образом, `T&&` будет `int& &&`, что сворачивается в `int&`. Таким образом, даже при том, что формой параметра функции `f3()` будет ссылка на `r`-значение (т.е. `T&&`), этот вызов создаст экземпляр функции `f3()` с типом ссылки на `l`-значение (т.е. `int&`):

`void f3<int&>(int&); // когда T — это int&, параметр функции`

// сворачивается в `int&`

У этих правил есть два важных следствия.

- Параметр функции, являющийся ссылкой на `r`-значение для параметра типа шаблона (например, `T&&`), может быть связан с `l`-значением.

- Если аргумент будет `l`-значением, то выведенный тип аргумента шаблона будет типом ссылки на `l`-значение, и экземпляр параметра функции будет создан как (обычный) параметр ссылки на `l`-значение (`T&`).

Стоит также обратить внимание на то, что параметру функции `T&&` косвенно можно передать аргумент любого типа. Параметр такого типа может использоваться с `r`-значениями, а, как было продемонстрировано только что, также и с `l`-значениями.



Параметру функции, являющемуся ссылкой на `r`-значение на тип параметра шаблона (т.е. `T&&`), может быть передан аргумент любого типа. Когда такому параметру передается `l`-значение, экземпляр параметра функции создается как обычная ссылка на `l`-значение (`T&`).

Шаблоны функций с параметрами ссылки на r-значения

У того факта, что параметр шаблона может быть выведен как ссылочный тип, имеются удивительные последствия для кода в шаблоне:

```
template <typename T> void f3(T&& val) {  
    T t = val; // копировать или привязать ссылку?  
    t = fcn(t); // изменит ли присвоение только t или  
    val и t?  
    if (val == t) { /* ... */ } // всегда истинно,  
если T — ссылочный тип
```

```
}
```

Когда вызов функции `f3()` происходит для такого `r`-значения, как литерал `42`, `T` имеет тип `int`. В данном случае локальная переменная `t` имеет тип `int` и инициализируется при копировании значения параметра `val`. При присвоении переменной `t` параметр `val` остается неизменным.

С другой стороны, когда происходит вызов функции `f3()` для `l`-значения `i`, типом `T` будет `int&`. Когда определяется и инициализируется локальная переменная `t`, у нее будет тип `int&`. Инициализация переменной `t` связывает ее с параметром `val`. При присвоении переменной `t` одновременно изменяется и параметр `val`. В этом экземпляре функции `f3()` оператор `if` всегда будет возвращать значение `true`.

На удивление сложно написать правильный код, когда задействованные типы могут быть простыми (не ссылочными) типами или ссылочными типами (хотя такие классы трансформации типов, как `remove_reference` (см. раздел 16.2.3), вполне могут помочь в этом).

На практике параметры в виде ссылки на `r`-значение используются в одном из двух случаев: либо когда шаблон перенаправляет свои аргументы, либо когда шаблон перегружается. Перенаправление рассматривается в разделе 16.2.7, а перегрузка шаблона в разделе 16.3, а пока достаточно знать, что стоит обратить внимание на то, что шаблоны функций, использующие ссылки на `r`-значение, зачастую используют перегрузку таким же образом, как описано в разделе 13.6.3:

```
template <typename T> void f( T&& );           // привязка
к не константным
                                                // r-
значениям
template <typename T> void f( const T& );    // l-
значения и константные
                                                // r-
значения
```

Подобно нешаблонным функциям, первая версия будет связана с изменяемым `r`-значением, а вторая с `l`-значением или константным `r`-значением.

Упражнения раздела 16.2.5

Упражнение 16.42. Определите типы `T` и `val` в каждом из следующих вызовов:

```
template <typename T> void g( T&& val );
```

```
int i = 0; const int ci = i;
(a) g(i); (b) g(ci); (c) g(i * ci);
```

Упражнение 16.43. Используя определенную в предыдущем упражнении функцию, укажите, каким будет параметр шаблона `g()` при вызове `g(i = ci)`?

Упражнение 16.44. Используя те же три вызова, что и в первом упражнении, определите типы `T`, если параметр функции `g()` объявляется как `T` (а не `T&&`) и как `const T&`?

Упражнение 16.45. С учетом следующего шаблона объясните происходящее при вызове функции `g()` с таким литеральным значением, как `42`, и с переменной типа `int`?

```
template <typename T> void g(T&& val) { vector<T>
v; }
```



16.2.6. Функция `std::move()`

Библиотечная функция `move()` (см. раздел 13.6.1) — хороший пример шаблона, использующего ссылки на `r`-значение. К счастью, функцию `move()` можно использовать, не понимая механизма работы используемого ею шаблона. Однако изучение работы функции `move()` может помочь понять и использовать шаблоны.

В разделе 13.6.2 обращалось внимание на то, что, хотя и нельзя непосредственно привязать ссылку на `r`-значение к `l`-значению, функцию `move()` можно использовать для получения ссылки на `r`-значение, связанной с `l`-значением. Поскольку функция `move()` может получать аргументы, по существу, любого типа, нет ничего удивительного в том, что `move()` — это шаблон функции.

Как определена функция `std::move()`?

Стандартное определение функции `move()` таково:

```
// об использовании typename в типе возвращаемого
значения и
// приведении см. раздел 16.1.3
// remove_reference рассматривается в разделе
16.2.3
template <typename T>
```

```

typename remove_reference<T>::type&& move( T&& t) {
    // static_cast рассматривается в разделе 4.11.3
    return static_cast<typename
remove_reference<T>::type&&>( t);
}

```

Этот код короток, но сложен. В первую очередь, параметр функции `move()`, `T&&` является ссылкой на `r`-значение типа параметра шаблона. Благодаря сворачиванию ссылок этот параметр может соответствовать аргументу любого типа. В частности, функции `move()` можно передать либо `l`-, либо `r`-значение:

```

string s1( "hi! " ), s2;
s2 = std::move( string( "bye! " ) ); // ok: перемещение
r-значения
s2 = std::move( s1 ); // ok: но после присвоения
// значение s1 неопределено

```

Как работает функция `std::move()`

В первом присвоении аргумент функции `move()` является `r`-значением, полученным в результате выполнения конструктора `string("bye")` класса `string`. Как уже упоминалось, при передаче `r`-значения ссылочному `r`-значению параметра функции выведенный из этого аргумента тип является ссылочным типом (см. раздел 16.2.5). Таким образом, в вызове `std::move(string("bye! "))`:

- выведенным типом `T` будет `string`;
- следовательно, экземпляр шаблона `remove_reference` создается с типом `string`;
- тип-член `type` класса `remove_reference<string>` будет иметь тип `string`;
- типом возвращаемого значения функции `move()` будет `string&&`;
- у параметра `t` функции `move()` будет тип `string&&`;

Соответственно, этот вызов создает экземпляр `move<string>`, являющийся следующей функцией:

```
string&& move( string &&t)
```

Тело этой функции возвращает тип `static_cast<string&&>(t)`. Типом `t` уже является `string&&`, поэтому приведение не делает ничего. Следовательно, результатом этого вызова будет ссылка на `r`-значение, которое было дано.

Теперь рассмотрим второе присвоение, которое вызывает функцию

`std::move(s1)`. В этом вызове аргументом функции `move()` является l-значение. Поэтому на сей раз:

- выведенным типом `T` будет `string&` (ссылка на тип `string`, а не просто `string`);
- следовательно, экземпляр шаблона `remove_reference` создается с типом `string&`;
- тип-член `type` класса `remove_reference<string&>` будет иметь тип `string`;
- типом возвращаемого значения функции `move()` все еще будет `string&&`;
- параметр `t` функции `move()` будет создан как экземпляр `string&&`, который сворачивается в `string&`.

Таким образом, этот вызов создает экземпляр шаблона `move<string&>`, который является точно тем, что необходимо для связи ссылки на r-значение с l-значением.

```
string&& move(string &t)
```

Тело этого экземпляра возвращает тип `static_cast<string&&>(t)`. В данном случае типом `t` является `string&`, который приведение преобразует в тип `string&&`.

Оператор `static_cast` поддерживает приведение l-значения к ссылке на r-значение



Обычно оператор `static_cast` может выполнить только доступные преобразования (см. раздел 16.3). Однако для ссылок на r-значение есть специальное разрешение: даже при том, что нельзя явно преобразовать l-значение в ссылку на r-значение, используя оператор `static_cast`, можно явно привести l-значение к ссылке на r-значение.

Привязка ссылки на r-значение к l-значению создает код, который работает с разрешением ссылке на r-значение заменять l-значение. Иногда, как в случае с функцией `reallocate()` класса `StrVec` (см. раздел 13.6.1), известно, что замена l-значения безопасна. Разрешая осуществлять это приведение, язык позволяет его использование. *Вынуждая* использовать приведение, язык пытается предотвратить его случайное использование.

И наконец, хотя такие приведения можно написать непосредственно,

намного проще использовать библиотечную функцию `move()`. Кроме того, использование функции `std::move()` существенно облегчает поиск в коде места, потенциально способного заменить l-значения.

Упражнения раздела 16.2.6

Упражнение 16.46. Объясните, что делает этот цикл из функции `StrVec::reallocate()` (раздел 13.5):

```
for (size_t i = 0; i != size(); ++i)
    alloc.construct(dest++, std::move(*elem++));
```



16.2.7. Перенаправление

Некоторые функции должны перенаправлять другим функциям один или несколько своих аргументов с *неизменными* типами. В таких случаях необходимо сохранить всю информацию о перенаправленных аргументах, включая то, является ли тип аргумента константой и является ли аргумент l- или r-значением.

В качестве примера напишем функцию, получающую вызываемое выражение и два дополнительных аргумента. Функция вызовет предоставленное вызываемое выражение с другими двумя аргументами в обратном порядке. Вот первый фрагмент функции обращения:

```
// шаблон, получающий вызываемое выражение и два
параметра
// вызывает предоставленное выражение с
// "обращенными" параметрами
// flip1 - неполная реализация: спецификатор const
верхнего уровня и
// ссылки теряются
template <typename F, typename T1, typename T2>
void flip1(F f, T1 t1, T2 t2) {
    f(t2, t1);
}
```

Этот шаблон работает прекрасно, пока он не используется для вызова функции со ссылочным параметром:

```
void f(int v1, int &v2) // обратите внимание, v2 -
ссылка
{
```

```
    cout << v1 << " " << ++v2 << endl;
}
```

Здесь функция `f()` изменяет значение аргумента, привязанного к параметру `v2`. Но если происходит вызов функции `f()` через шаблон `flip1`, внесенные функцией `f()` изменения не затронут первоначальный аргумент:

```
f( 42, i);           // f() изменяет свой аргумент i
flip1( f, j, 42);   // вызов f() через flip1 оставляет
j неизменным
```

Проблема в том, что `j` передается параметру `t1` шаблона `flip1`. Этот параметр имеет простой, не ссылочный тип `int`, а не `int&`. Таким образом, этот вызов создает следующий экземпляр шаблона `flip1`:

```
void flip1( void( *fcn)( int, int& ), int t1, int t2 );
```

Значение `j` копируется в `t1`. Ссылочный параметр в функции `f()` связан с `t1`, а не с `j`.



Определение параметров функции, хранящих информацию типа

Чтобы передать ссылку через функцию, необходимо переписать ее так, чтобы параметры сохраняли принадлежность своих аргументов к l-значениям. Немного поразмыслив, можно предположить, что константность аргументов также необходимо сохранить.

Всю информацию о типе аргумента можно сохранить, определив соответствующий ему параметр функции как ссылку на r-значение параметра типа шаблона. Использование ссылочного параметра (l- или r-значение) позволяет сохранить константность, поскольку спецификатор `const` в ссылочном типе нижнего уровня. Благодаря сворачиванию ссылок (см. раздел 16.2.5), если определить параметры функции как `T1&&` и `T2&&`, можно сохранить принадлежность к l- или r-значениям аргументов функции (см. раздел 16.2.5):

```
template <typename F, typename T1, typename T2>
void flip2( F f, T1 &&t1, T2 &&t2 ) {
    f( t2, t1 );
}
```

Как и прежде, если происходит вызов `flip2(f, j, 42)`, l-значение `j` передается параметру `t1`. Однако в функции `flip()` для `T1` выводится тип `int&`, а значит, тип `t1` сворачивается в `int&`. Ссылка `t1` связана с `j`.

Когда функция `flip()` вызывает функцию `f()`, ссылочный параметр `v2` в функции `f()` привязан к `t1`, который, в свою очередь, привязан к `j`. Когда функция `f()` осуществляет инкремент `v2`, это изменяет значение `j`.



Параметр функции, являющийся ссылкой на `r`-значение параметра типа шаблона (т.е. `T&&`), сохраняет константность и принадлежность к `l`- или `r`-значениям соответствующих ему аргументов.

Эта версия функции `flip()` решает одну половину проблемы. Она работает прекрасно с функциями, получающими ссылки на `l`-значение, но неприменима для вызова функций с параметрами ссылок на `r`-значение. Например:

```
void g(int &&i, int& j) {  
    cout << i << " " << j << endl;  
}
```

Если попытаться вызывать функцию `g()` через функцию `flip()`, то для параметра ссылки на `r`-значение функции `g()` будет передан параметр `t2`. Даже если функции `flip()` было передано `r`-значение, функции `g()` будет передан параметр, носящий в функции `flip()` имя `t2`:

```
flip2(g, i, 42); // ошибка: нельзя инициализировать  
int&& из l-значения
```

Параметр функции, как и любая другая переменная, является выражением `l`-значения (см. раздел 13.6.1). В результате вызов функции `g()` в функции `flip()` передает `l`-значение параметру ссылки на `r`-значение функции `g()`.



Использование функции `std::forward()` для сохранения информации типа в вызове



Чтобы передать функции `flip()` параметры способом, сохраняющим типы первоначальных аргументов, можно использовать новую библиотечную функцию `forward()`. Как и функция `move()`, функция

`forward()` определяется в заголовке `utility`. В отличие от функции `move()`, функцию `forward()` следует вызывать с явным аргументом шаблона (см. раздел 16.2.2). Для этого явного аргумента типа функция `forward()` возвращает ссылку на `r`-значение. Таким образом, типом возвращаемого значения функции `forward<T>` будет `T&&`.

Обычно функцию `forward()` используют для передачи параметра функции, который определен как ссылка на `r`-значение, параметру типа шаблона. Благодаря сворачиванию ссылок для типа возвращаемого значения функция `forward()` сохраняет характер (`l`- или `r`-значение) переданного ей аргумента:

```
template <typename Type> intermediary(Type &&arg) {  
    finalFcn(std::forward<Type>(arg)); // ...  
}
```

Здесь `Type` используется как тип явного аргумента шаблона функции `forward()` (выводимый из `arg`). Поскольку `arg` — это ссылка на `r`-значение для параметра типа шаблона, параметр `Type` представит всю информацию типа в аргументе, переданном параметру `arg`. Если этот аргумент будет `r`-значением, то параметр `Type` будет иметь обычный (не ссылочный) тип и функция `forward<Type>()` возвратит `Type&&`. Если аргумент будет `l`-значением, то (благодаря сворачиванию ссылок) типом параметра `Type` будет ссылка на `l`-значение. В данном случае типом возвращаемого значения будет ссылка на `r`-значение для типа ссылки на `l`-значение. Снова благодаря сворачиванию ссылок (на сей раз для типа возвращаемого значения) функция `forward<Type>()` возвратит тип ссылки на `l`-значение.



При использовании с параметром функции, являющимся ссылкой на `r`-значение для параметра типа шаблона (`T&&`), функция `forward()` сохраняет все подробности типа аргумента.

Перепишем первоначальную функцию, используя на этот раз функцию `forward()`:

```
template <typename F, typename T1, typename T2>  
void flip(F f, T1 &&t1, T2 &&t2) {
```

```
f( std::forward<T2>( t2 ), std::forward<T1>( t1 ) );  
}
```

Если происходит вызов функции `flip(g, i, 42)`, то параметр `i` будет передан функции `g()`, поскольку `int&` и `42` будут переданы как `int&&`.



Подобно функции `std::move()`, для функции `std::forward()` не стоит предоставлять объявление `using`. Причина описана в разделе 18.2.3.

Упражнения раздела 16.2.7

Упражнение 16.47. Напишите собственную версию функции обращения и проверьте ее, вызывая функции с параметрами ссылок на `r`-значение и `l`-значение.



16.3. Перегрузка и шаблоны

Шаблоны функций могут быть перегружены другими шаблонами или обычными, не шаблонными функциями. Как обычно, функция с тем же именем должна отличаться либо количеством, либо типом своих параметров.

На подбор функции (см. раздел 6.4) присутствие шаблона функции влияет следующими способами.

- В набор функций-кандидатов на вызов включаются любые экземпляры шаблона функции, для которой успешна дедукция аргумента шаблона (см. раздел 16.2).
- Шаблоны функций-кандидатов всегда подходящие, поскольку дедукция аргумента шаблона устранит все неподходящие шаблоны.
- Как обычно, подходящие функции (шаблонные и нешаблонные) ранжируются по преобразованиям, если таковые вообще имеются. Конечно, набор применимых преобразований при вызове шаблона функции весьма ограничен (см. раздел 16.2.1).
 - Так же как обычно, если только одна функция обеспечивает наилучшее соответствие, она и выбирается. Но если одинаково хорошее соответствие обеспечивают несколько функций, то:
 - если в наборе одинаково хороших соответствий есть только одна нешаблонная функция, то выбрана будет она;
 - если в наборе нет нешаблонных функций, но есть несколько шаблонных, и одна из них более специализированна, чем любые другие, то будет выбран более специализированный шаблон функции;
 - в противном случае вызов неоднозначен.



Правильное определение набора перегруженных шаблонов функций требует хорошего понимания отношений между типами и ограничений на преобразования, применимых к аргументам в шаблонах функций.

Создание перегруженных шаблонов

В качестве примера создадим набор функций, которые могли бы

пригодиться во время отладки. Назовем отладочные функции `debug_rep()`, каждая из них возвратит строковое представление предоставленного объекта. Начнем с создания самой общей версии этой функции в качестве шаблона, получающего ссылку на константный объект:

```
// выводит любом тип, который иначе не обработать
template <typename T> string debug_rep(const T &t)
{
    ostringstream ret; // см. раздел 8.3
    ret << t; // использует оператор вывода T для
    // вывода представления t
    return ret.str(); // возвращает копию строки, с
    // которой связан ret
}
```

Эта функция применяется для создания строки, соответствующей объекту любого типа, у которого есть оператор вывода.

Теперь определим версию функции `debug_rep()` для вывода указателя:

```
// выводит указатели как их значение,
// сопровожданное объектом,
// на который он указывает
// обратите внимание: эта функция не будет работать
// правильно с char*;
// см. раздел 16.3
template <typename T> string debug_rep(T *p) {
    ostringstream ret;
    ret << "pointer: " << p; // выводит собственное
    // значение указателя
    if (p)
        ret << " " << debug_rep(*p); // выводит значение,
    // на которое
    // указывает p
    else
        ret << " null pointer"; // или указывает, что p -
    // нулевой
    return ret.str(); // возвращает копию строки, с
    // которой связан ret
}
```

Эта версия создает строку, содержащую собственное значение указателя и вызывает функцию `debug_rep()` для вывода объекта, на

который указывает этот указатель. Обратите внимание, что эта функция не может использоваться для вывода символьных указателей, поскольку библиотека ввода-вывода определяет версию оператора << для значения указателя `char*`. Эта версия оператора << подразумевала, что указатель обозначает символьный массив с нулевым символом в конце и выводит содержимое массива, а не его адрес. Обработка символьных указателей рассматривается в разделе 16.3.

Эти функции можно использовать следующим образом:

```
string s("hi");
cout << debug_rep(s) << endl;
```

Подходящей для этого вызова является только первая версия функции `debug_rep()`. Второй версии требуется параметр в виде указателя, а в этом вызове передан не указатель. Нет никакого способа создать экземпляр шаблона функции, ожидающего тип указателя, из параметра, который не является указателем, поэтому дедукция аргумента терпит неудачу. Поскольку есть только одна подходящая функция, она и используется.

Если происходит вызов функции `debug_rep()` с указателем:

```
cout << debug_rep(&s) << endl;
```

то обе функции создают подходящие экземпляры:

- `debug_rep(const string*&)` — экземпляр первой версии функции `debug_rep()` с привязкой параметра `T` к типу `string*`;
- `debug_rep(string*)` — экземпляр второй версии функции `debug_rep()` с привязкой параметра `T` к типу `string`.

Точным соответствием для этого вызова является экземпляр второй версии функции `debug_rep()`. Создание экземпляра первой версии требует преобразования простого указателя в указатель на константу. Обычный подбор функции гласит, что следует предпочесть второй шаблон, и в действительности так и происходит.

Несколько подходящих шаблонов

В качестве другого примера рассмотрим следующий вызов:

```
const string *sp = &s;
cout << debug_rep(sp) << endl;
```

Здесь подходящими являются оба шаблона, и оба обеспечивают точное соответствие:

- `debug_rep(const string*&)` — экземпляр первой версии шаблона с привязкой параметра `T` к типу `const string*`;
- `debug_rep(const string*)` — экземпляр второй версии

шаблона с привязкой параметра T к типу `const string`.

В данном случае обычный подбор функции не может различить эти два вызова. Можно было бы ожидать, что этот вызов будет неоднозначен. Однако благодаря специальному правилу для перегруженных шаблонов функций этот вызов решается как `debug_rep(T*)`, поскольку это более специализированный шаблон.

Причина для этого правила в том, что без него не было бы никакого способа вызвать версию функции `debug_rep()` для указателя на константу. Проблема в том, что к шаблону `debug_rep(const T&)` подходит практически любой тип, включая типы указателя. Этот шаблон является более общим, чем `debug_rep(T*)`, который может быть вызван только для типов указателя. Без этого правила вызовы с передачей указателей на константу всегда будут неоднозначны.



Когда есть несколько перегруженных шаблонов, предоставляющих одинаково хорошее соответствие для вызова, предпочтается наиболее специализированная версия.

Не шаблон и перегрузка шаблона

Для следующего примера определим обычную, не шаблонную версию функции `debug_rep()`, выводящую строки в двойных кавычках:

```
// вывод строк в двойных кавычках
string debug_rep( const string &s ) {
    return '"' + s + '"';
}
```

Теперь, когда происходит вызов функции `debug_rep()` для строки:

```
string s("hi");
cout << debug_rep(s) << endl;
```

есть две одинаково хорошо подходящих функции:

- `debug_rep<string>(const string&)` — первый шаблон с привязкой параметра T к типу `string`;
- `debug_rep(const string&)` — обычная, не шаблонная функция.

В данном случае у обеих функций одинаковый список параметров, поэтому каждая из них обеспечивает одинаково хорошее соответствие этому вызову. Однако выбирается нешаблонная версия. По тем же

причинам, по которым предпочтитаются наиболее специализированные из одинаково хорошо подходящих шаблонов функций, нешаблонная функция предпочтается при одинаково хорошем соответствии с шаблонной функцией.



Когда нешаблонная функция обеспечивает одинаково хорошее соответствие с шаблонной функцией, предпочтается нешаблонная версия.

Перегруженные шаблоны и преобразования

До сих пор не рассматривался случай с указателями на символьные строки в стиле С и строковые литералы. Теперь, когда имеется версия функции `debug_rep()`, получающая строку, можно было бы ожидать, что ей будет соответствовать вызов, которому переданы символьные строки. Однако рассмотрим этот вызов:

```
cout << debug_rep("hi world!") << endl; // вызов  
debug_rep(T*)
```

Здесь подходящими являются все три функции `debug_rep()`:

- `debug_rep(const T&)` — с привязкой параметра `T` к типу `char[10]`;
- `debug_rep(T*)` — с привязкой параметра `T` к типу `const char`;
- `debug_rep(const string&)` — требующая преобразования из `const char*`` `bstring`.

Оба шаблона обеспечивают точное соответствие аргументу — второй шаблон требует (допустимого) преобразования из массива в указатель, и это преобразование считается точным соответствием при подборе функции (см. раздел 6.6.1). Нешаблонная версия является подходящей, но требует пользовательского преобразования. Эта функция хуже точного соответствия, поэтому кандидатами остаются два шаблона. Как и прежде, версия `T*` более специализирована, она и будет выбрана.

Если символьные указатели необходимо обработать как строки, можно определить еще две перегруженные, нешаблонные функции:

```
// преобразовать символьные указатели в строку и  
// вызвать строковую  
// версию debug_rep()  
string debug_rep(char *p) {
```

```

    return debug_rep( string( p ) );
}
string debug_rep( const char *p) {
    return debug_rep( string( p ) );
}

```

Пропуск объявления может нарушить программу

Следует заметить, что для правильной работы версии `char*` функции `debug_rep()` объявление `debug_rep(const string&)` должно находиться в области видимости, когда эти функции определяются. В противном случае будет вызвана неправильная версия функции `debug_rep()`:

```

template <typename T> string debug_rep( const T &t);
template <typename T> string debug_rep( T *p);
// следующее объявление должно быть в области
// видимости
// для правильного определения debug_rep( char * )
string debug_rep( const string & );
string debug_rep( char *p) {
    // если объявление для версии, получающей const
    string&, не находится
    // в области видимости, return вызовет call
    debug_rep( const T& ) с
    // экземпляром строки в параметре T
    return debug_rep( string( p ) );
}

```

Обычно, если попытаться использовать функцию, которую забыли объявлять, код не будет откомпилирован. Но с функциями, которые перегружают шаблон функции, все не так. Если компилятор может создать экземпляр вызова из шаблона, то отсутствие объявления не будет иметь значения. В этом примере, если забыть объявлять версию функции `debug_rep()`, получающую строку, компилятор *тихо* создаст версию экземпляра шаблона, получающую `const T&`.



Объявляйте каждую функцию в наборе перегруженных, прежде чем определять их. Таким образом можно гарантировать, что компилятор

создаст экземпляр вызова прежде, чем он встретит функцию, которую предполагалось вызвать.

Упражнения раздела 16.3

Упражнение 16.48. Напишите собственные версии функций `debug_rep()`.

Упражнение 16.49. Объясните, что происходит в каждом из следующих вызовов:

```
template <typename T> void f( T );
template <typename T> void f( const T* );
template <typename T> void g( T );
template <typename T> void g( T* );
int i = 42, *p = &i;
const int ci = 0, *p2 = &ci;
g( 42 ); g( p ); g( ci ); g( p2 );
f( 42 ); f( p ); f( ci ); f( p2 );
```

Упражнение 16.50. Определите функции из предыдущего упражнения так, чтобы они выводили идентификационное сообщение. Выполните код этого упражнения. Если вызовы ведут себя не так, как ожидалось, выясните почему.



16.4. Шаблоны с переменным количеством аргументов



Шаблон с переменным количеством аргументов (variadic template) — это шаблон функции или класса, способный получать переменное количество параметров. Набор таких параметров называется *пакетом параметров* (parameter pack). Есть два вида пакетов параметров: *пакет параметров шаблона* (template parameter pack), представляющий любое количество параметров шаблона, и *пакет параметров функции* (function parameter pack), представляющий любое количество параметров функции.

Для указания, что шаблону или функции представлен пакет параметров, используется многоточие. В списке параметров шаблона синтаксис `class...` или `typename...` означает, что следующий параметр представляет список любого количества типов; имя типа, сопровождаемое многоточием, представляет список из любого количества параметров значения заданного типа. Параметр в списке параметров функции, типом которого является пакет параметров шаблона, представляет собой пакет параметров функции. Например:

```
// Args - это пакет параметров шаблона; rest -  
пакет параметров функции  
// Args представляет любое количество параметров  
типа шаблона  
// rest представляет любое количество параметров  
функции
```

```
template <typename T, typename... Args>  
void foo(const T &t, const Args& ... rest);
```

Этот код объявляет, что `foo()` — это функция с переменным количеством аргументов, у которой один параметр типа по имени `T` и пакет параметров шаблона по имени `Args`. Этот пакет представляет любое количество дополнительных параметров типа. В списке параметров функции `foo()` один параметр типа `const&` для любого типа переданного параметром `T` и пакет параметров функции `rest`. Этот пакет представляет любое количество параметров функции.

Как обычно, компилятор выводит типы параметра шаблона из

аргументов функции. Для шаблона с переменным количеством аргументов компилятор также выводит количество параметров в пакете. Рассмотрим, например, следующие вызовы:

```
int i = 0; double d = 3.14; string s = "how now  
brown cow";
```

```
foo(i, s, 42, d); // три параметра в пакете  
foo(s, 42, "hi"); // два параметра в пакете  
foo(d, s); // один параметр в пакете  
foo("hi"); // пустой пакет
```

Компилятор создаст четыре разных экземпляра функции `foo()`:

```
void foo(const int&, const string&, const int&,  
const double&);  
void foo(const string&, const int&, const  
char[3] &);  
void foo(const double&, const string&);  
void foo(const char[3] &);
```

В каждом случае тип Т выводится из типа первого аргумента. Остальные аргументы (если они есть) представляют количество и типы дополнительных аргументов функции.

Оператор `sizeof...`



Когда необходимо узнать, сколько элементов находится в пакете, можно использовать *оператор `sizeof...`*. Как и оператор `sizeof` (см. раздел 4.9), оператор `sizeof...` возвращает константное выражение (см. раздел 2.4.4) и не вычисляет свой аргумент:

```
template<typename ... Args> void g(Args ... args) {  
    cout << sizeof...(Args) << endl; // количество  
    // параметров типа  
    cout << sizeof...(args) << endl; // количество  
    // параметров функции  
}
```

Упражнения раздела 16.4

Упражнение 16.51. Определите, что возвратят операторы `sizeof...` (`Args`) и `sizeof...(rest)` для каждого вызова функции `foo()` в этом разделе.

Упражнение 16.52. Напишите программу, проверяющую ответы на предыдущий вопрос.



16.4.1. Шаблоны функции с переменным количеством аргументов

В разделе 6.2.6 упоминалось, что для определения функции, способной получать переменное количество аргументов, можно использовать класс `initializer_list`. Однако у аргументов должен быть одинаковый тип (или типы, преобразуемые в общий тип). Функции с переменным количеством аргументов используются тогда, когда не известно ни количество, ни типы аргументов. Для примера определим функцию, подобную прежней функции `error_msg()`, только на сей раз обеспечим и изменение типов аргумента. Начнем с определения функции `print()` с переменным количеством аргументов, которая выводит содержимое заданного списка аргументов в указанный поток.

Функции с переменным количеством аргументов зачастую рекурсивны (см. раздел 6.3.2). Первый вызов обрабатывает первый аргумент в пакете и вызывает себя для остальных аргументов. Новая функция `print()` будет работать таким же образом — каждый вызов выводит свой второй аргумент в поток, обозначенный первым аргументом. Для остановки рекурсии следует определить также обычную функцию `print()`, которая получает поток и объект:

```
// Функция для завершения рекурсии и вывода
последнего элемента
// ее следует объявить перед определением версией
print() с переменным
// количеством аргументов
template<typename T>
ostream &print(ostream &os, const T &t) {
    return os << t; // нет разделителя после
последнего элемента в пакете
}
// эта версия print() будет вызвана для всех
элементов в пакете, кроме
// последнего
```

```

template <typename T, typename... Args>
ostream &print(ostream &os, const T &t, const
Args&... rest) {
    os << t << ", "; // выводит первый аргумент
    return print(os, rest...); // рекурсивный вызов;
вывод других
                                // аргументов
}

```

Первая версия функции `print()` останавливает рекурсию и выводит последний аргумент в начальном вызове функции `print()`. Вторая версия, с переменным количеством аргументов, выводит аргумент, связанный с `t`, и вызывает себя для вывода остальных значений в пакете параметров функции.

Ключевая часть — вызов функции `print()` в функции с переменным количеством аргументов:

```

return print(os, rest...); // рекурсивный вызов;
вывод других
                                // аргументов

```

Версия функции `print()` с переменным количеством аргументов получает три параметра: `ostream&`, `const T&` и пакет параметров. Но в этом вызове передаются только два аргумента. В результате первый аргумент в пакете `rest` привязывается к `t`. Остальные аргументы в пакете `rest` формируют пакет параметров для следующего вызова функции `print()`. Таким образом, при каждом вызове первый аргумент удаляется из пакета и становится аргументом, связанным с `t`. Соответственно, получаем:

```
print(cout, i, s, 42); // два параметра в пакете
```

Рекурсия выполнится следующим образом:

Вызов	<code>t</code>	<code>rest...</code>
<code>print(cout, i, s, 42)</code>	<code>i</code>	<code>s, 42</code>
<code>print(cout, s, 42)</code>	<code>s</code>	<code>42</code>

Вызов `print(cout, 42)` вызывает обычную версию функции `print()`.

Первые два вызова могут соответствовать только версии функции `print()` с переменным количеством аргументов, поскольку обычная версия не является подходящей. Эти вызовы передают четыре и три аргумента соответственно, а обычная функция `print()` получает только

два аргумента.

Для последнего вызова в рекурсии, `print(cout, 42)`, подходят обе версии функции `print()`. Этот вызов передает два аргумента, и типом первого являются `ostream&`. Таким образом, подходящей является обычная версия функции `print()`.

Версия с переменным количеством аргументов также является подходящей. В отличие от обычного аргумента, пакет параметров может быть пустым. Следовательно, экземпляр версии функции `print()` с переменным количеством аргументов может быть создан только с двумя параметрами: один — для параметра `ostream&` и другой — для параметра `const T&`.

Обе функции обеспечивают одинаково хорошее соответствие для вызова. Однако нешаблонная версия с переменным количеством аргументов более специализирована, чем шаблонная с переменным количеством аргументов. Поэтому выбирается версия без переменного количества аргументов (см. раздел 16.3).



Объявление версии функции `print()` с постоянным количеством аргументов должно быть в области видимости, когда определяется версия с переменным количеством аргументов. В противном случае функция с переменным количеством аргументов будет рекурсивно вызывать себя бесконечно.

Упражнения раздела 16.4.1

Упражнение 16.53. Напишите собственные версии функций `print()` и проверьте их, выводя один, два и пять аргументов, у которых должны быть разные типы.

Упражнение 16.54. Что происходит при вызове функции `print()` для типа, не имеющего оператора `<<?`

Упражнение 16.55. Объясните, как выполнилась бы версия функции `print()` с переменным количеством аргументов, если бы обычная версия функции `print()` была объявлена после определения версии с переменным количеством аргументов.



16.4.2. Разворачивание пакета

Кроме выяснения размера, единственное, что можно еще сделать с пакетом параметров, — это *развернуть* (pack expansion) его. При развертывании пакета предоставляется *схема* (pattern), используемая для каждого развернутого элемента. Разворачивание пакета разделяет его на элементы с применением схемы к каждому из них. Для запуска развертывания справа от схемы помещают многоточие (...).

Например, функция print() содержит два развертывания:

```
template <typename T, typename... Args> ostream &
print(ostream &os, const T &t, const Args&... rest)
// развертывание

// Args
{
    os << t << ", ";
    return print(os, rest...); // развертывание rest
}
```

В первом случае развертывание пакета параметров шаблона создает список параметров функции print(). Второй случай развертывания находится в вызове функции print(). Эта схема создает список аргументов для вызова.

Разворачивание пакета Args применяет схему const Args& к каждому элементу в пакете параметров шаблона Args. Результатом этой схемы будет разделенный запятыми список из любого количества типов параметров в формате const *тип*&. Например:

```
print(cout, i, s, 42); // два параметра в пакете
```

Типы последних двух аргументов, наряду со схемой, определяют типы замыкающих параметров. Этот вызов создает следующий экземпляр:

```
ostream&
print(ostream&, const int&, const strings, const
int&);
```

Второе развертывание происходит в рекурсивном вызове функции print(). В данном случае схема — это имя пакета параметров функции (т.е. rest). Эта схема разворачивается в разделяемый запятыми список элементов пакета. Таким образом, этот вызов эквивалентен следующему:

```
print(os, s, 42);
```

Концепция развертывания пакета

Развертывание пакета параметров функции `print()` только разворачивало пакет на его составные части. При развертывании пакета параметров функции возможны и более сложные схемы. Например, можно было бы написать вторую функцию с переменным количеством аргументов, которая вызывает функцию `debug_rep()` (см. раздел 16.3) для каждого из своих аргументов, а затем вызывает функцию `print()`, чтобы вывести полученные строки:

```
// вызвать debug_rep() для каждого аргумента в
// вызове print()
template <typename... Args>
ostream &errorMsg( ostream &os, const Args&... rest)
{
    // print(os, debug_rep(a1), debug_rep(a2), ...,
    debug_rep(an)
    return print(os, debug_rep(rest)...);
}
```

Вызов функции `print()` использует схему `debug_rep(rest)`. Эта схема означает, что функцию `debug_rep()` следует вызвать для каждого элемента в пакете параметров функции `rest`. Получившийся развернутый пакет будет разделяемым запятыми списком вызовов функции `debug_rep()`. Таким образом, вызов

```
errorMsg( cerr, fcnName, code.num(), otherData,
"other", item);
```

выполняется, как будто было написано:

```
print( cerr, debug_rep( fcnName),
debug_rep( code.num() ),
debug_rep( otherData ),
debug_rep( "otherData" ),
debug_rep( item ) );
```

Следующая схема, напротив, не была бы откомпилирована:

```
// передает пакет debug_rep(); print(os,
debug_rep( a1, a2, an ))
print( os, debug_rep( rest... ) ); // ошибка: нет
функции, соответствующей
// вызову
```

Проблема здесь в том, что пакет `rest` развернут в вызове функции `debug_rep()`. Этот вызов выполнился бы так, как будто было написано:

```
print( cerr, debug_rep( fcnName, code.num( ),
```

```
otherData, "otherData", item);
```

В этом развертывании осуществляется попытка вызова функции `debug_rep()` со списком из пяти аргументов. Нет никакой версии функции `debug_rep()`, соответствующей этому вызову. Функция `debug_rep()` имеет постоянное количество аргументов, и нет никакой ее версии с пятью параметрами.



Схема при развертывании применяется по отдельности к каждому элементу в пакете.

Упражнения раздела 16.4.2

Упражнение 16.56. Напишите и проверьте версию функции `errorMsg()` с переменным количеством аргументов.

Упражнение 16.57. Сравните свою версию функции `errorMsg()` с переменным количеством аргументов с функцией `error_msg()` из раздела 6.2.6. Каковы преимущества и недостатки каждого подхода?



16.4.3. Перенаправление пакетов параметров



По новому стандарту можно использовать шаблоны с переменным количеством аргументов совместно с функцией `forward()` для написания функций, которые передают свои аргументы неизменными некой другой функции. Чтобы проиллюстрировать такие функции, добавим в класс `StrVec` (см. раздел 13.5) функцию-член `emplace_back()`. Такая функция-член библиотечных контейнеров является шаблоном-членом с переменным количеством аргументов (см. раздел 16.1.4), которая использует их для создания элементов непосредственно в области, управляемой контейнером.

Версия функции `emplace_back()` для класса `StrVec` также должна быть с переменным количеством аргументов, поскольку у класса `string` много конструкторов, которые отличаются своими параметрами.

Поскольку желательно быть в состоянии использовать конструктор перемещения класса `string`, необходимо будет также сохранять всю информацию о типах аргументов, переданных функции `emplace_back()`.

Как уже упоминалось, сохранение информации типа — двухступенчатый процесс. Во-первых, для сохранения информации типа аргументов параметры функции `emplace_back()` следует определить как ссылки на `r`-значение параметра типа шаблона (см. раздел 16.2.7):

```
class StrVec {  
public:  
    template <class... Args> void  
emplace_back(Args&&...);  
    // остальные члены, как в разделе 13.5  
};
```

Схема `&&` в развертывании пакета параметров шаблона означает, что каждый параметр функции будет ссылкой на `r`-значение на соответствующий ей аргумент.

Во-вторых, функцию `forward()` следует использовать для сохранения первоначальных типов аргументов, когда функция `emplace_back()` передает их функции `construct()` (см. раздел 16.2.7):

```
template <class... Args>  
inline  
void StrVec::emplace_back(Args&&... args) {  
    chk_n_alloc(); // пересоздает StrVec при  
необходимости  
    alloc.construct(first_free++, std::forward<Args>  
(args)...);  
}
```

Тело функции `emplace_back()` вызывает функцию `chk_n_alloc()` (см. раздел 13.5), чтобы гарантировать наличие достаточного места для элемента, и вызывает функцию `construct()`, чтобы создать элемент в позиции, на которую указывает указатель `first_free`.

```
std::forward<Args>(args)...
```

Развертывание в вызове функции `construct()` разворачивает оба пакета: параметров шаблона `Args` и параметров функции `args`. Эта схема создает элементы в формате:

```
std::forward<Tii )
```

где T_i представляет тип i -го элемента в пакете параметров шаблона, а t_i представляет i -й элемент в пакете параметров функции. Например, если `svec` имеет тип `StrVec`, то при вызове

```
svec.emplace_back( 10, 'c' ); // добавит cccccccccc  
как новый последний
```

// элемент

схема в вызове функции `construct()` развернется в

```
std::forward<int>( 10 ), std::forward<char>( c )
```

Использование функции `forward()` в этом вызове гарантирует, что если функция `emplace_back()` будет вызвана с r -значением, то функция `construct()` также получит r -значение. Например, в вызове

```
svec.emplace_back( s1 + s2 ); // использует  
конструктор перемещения
```

аргумент функции `emplace_back()` является r -значением, которое передается функции `construct()` как

```
std::forward<string>( string( "the end" ) )
```

Типом результата вызова `forward<string>` будет `strings&`, поэтому функция `construct()` будет вызвана со ссылкой на r -значение. Функция `construct()`, в свою очередь, перенаправит этот аргумент конструктору перемещения класса `string`, чтобы создать этот элемент.

Совет. Перенаправление и шаблоны с переменным количеством аргументов

Функции с переменным количеством аргументов зачастую перенаправляют свои параметры другим функциям. Форма таких функций, как правило, подобна функции `emplace_back()`:

```
// у функции fun() может быть любое количество  
параметров, каждый  
// из которых является ссылкой r-значения на тип  
параметра шаблона  
template<typename... Args>  
void fun( Args&&... args ) // развертывание Args в  
список ссылок  
// на r-значения  
{  
    // аргумент work() развертывает как Args, так и  
    args
```

```
    work( std::forward<Args>( args )... );  
}
```

Здесь предполагается перенаправить все аргументы функции `fun()` другой функции, `work()`, которая, по-видимому, осуществляет реальную работу. Как и вызов функции `construct()` в функции `emplace_back()`, развертывание в вызове функции `work()` разворачивает и пакет параметров шаблона, и пакет параметров функции.

Поскольку параметры функции `fun()` являются ссылками на `r`-значение, функции `fun()` можно передать аргументы любого типа; поскольку для передачи этих аргументов используется функция `std::forward()`, вся информация о типах этих аргументов будет сохраняться в вызове функции `work()`.

Упражнения раздела 16.4.3

Упражнение 16.58. Напишите функцию `emplace_back()` для собственного класса `StrVec` и для класса `Vec`, написанного в упражнении раздела 16.1.2.

Упражнение 16.59. С учетом того, что `s` имеет тип `string`, объясните вызов `svec.emplace_back(s)`.

Упражнение 16.60. Объясните, как работает функция `make_shared()` (см. раздел 12.1.1).

Упражнение 16.61. Определите собственную версию функции `make_shared()`.



16.5. Специализация шаблона

Не всегда можно написать один шаблон, который наилучшим образом подходит для всех возможных типов аргументов шаблона, для которых может быть создан его экземпляр. В некоторых случаях общий шаблон просто не подходит для типа: он либо приводит к ошибке при компиляции, либо к неправильным действиям. С другой стороны, иногда можно воспользоваться уникальными возможностями определенного типа для создания более эффективной функции, чем та, которой снабжен экземпляр общего шаблона.

Функция `compare()` — хороший пример шаблона функции, общее определение которого не подходит для специфического типа, а именно символьных указателей. Хотелось бы, чтобы функция `compare()` сравнивала символьные указатели, используя функцию `strcmp()`, а не сравнивала значения указателей. Действительно, ведь уже есть перегруженная функция `compare()`, обрабатывающая символьные строковые литералы (см. раздел 16.1.1):

```
// первая версия; может сравнить любые два типа
template <typename T> int compare(const T&, const
T&);
// вторая версия, для обработки строковых литералов
template<size_t N, size_t M>
int compare(const char (&)[N], const char (&)[M]);
```

Однако версия функции `compare()` с двумя параметрами значения шаблона будет вызвана только при передаче строкового литерала или массива. Если происходит вызов функции `compare()` с символьными указателями, будет вызвана первая версия шаблона:

```
const char *p1 = "hi", *p2 = "mom";
compare(p1, p2);           // вызывает первый шаблон
compare("hi", "mom");     // вызывает шаблон с двумя
параметрами значения
```

Нет никакого способа преобразовать указатель в ссылку на массив, поэтому вторая версия функции `compare()` не подходит для передачи указателей `p1` и `p2` как аргументов.

Для обработки символьных указателей (в отличие от массивов) можно определить *специализацию шаблона* (*template specialization*) для первой

версии функции `compare()`. Специализация — это отдельное определение шаблона, в котором определяется один или несколько параметров шаблона для получения специфического типа.

Специализация шаблона функции

При специализации шаблона функции следует предоставить аргументы для каждого параметра первоначального шаблона. Для указания специализации шаблона используется ключевое слово `template`, сопровожданное парой пустых угловых скобок (`<>`). Пустые скобки означают, что аргументы будут предоставлены для всех параметров первоначального шаблона:

```
// специальная версия compare() для работы с
указателями на символьные
// массивы
template <>
int compare(const char* const &p1, const char*
const &p2) {
    return strcmp(p1, p2);
}
```

Трудная для понимания часть этой специализации относится к типам параметра функции. При определении специализации типы параметров функции должны совпадать с соответствующими типами ранее объявленного шаблона:

```
template <typename T> int compare(const T&, const
T&);
```

В этой специализации параметры функции являются ссылками на константные типы. Подобно псевдонимам типа, взаимодействие между типами параметра шаблона, указателями и константами может удивить (см. раздел 2.5.1).

Необходимо определить специализацию шаблона этой функции с типом `const char*` для параметра `T`. Функция потребует ссылки на константную версию этого типа. Константная версия типа указателя — это константный указатель, а не указатель на константу (см. раздел 2.4.2). В данной специализации следует использовать тип `const char* const &`, являющийся ссылкой на константный указатель на константный символ.

Перегрузка функций или специализация шаблона

При определении специализации шаблона функции разработчик, по существу, выполняет задачу компилятора. Таким образом, определение

предоставляется для использования специфического экземпляра первоначального шаблона. Важно понимать, что специализация — это создание экземпляра функции; а не перегрузка ее экземпляра.



Специализация создает экземпляр шаблона, а не перегружает его. В результате специализация не затрагивает механизм подбора функций.

Может ли определение некой функции как специализации шаблона или как независимой, не шаблонной функции повлиять на подбор функций? Предположим, например, что имеется определение двух версий шаблонной функции `compare()`: той, что получает параметры как ссылки на массив, и другой, которая получает тип `const T&`. Факт наличия специализации для символьных указателей никак не влияет на подбор функции:

```
compare("hi", "tom")
```

Когда функция `compare()` вызывается для строкового литерала, оба шаблона функции оказываются подходящими и обеспечивают одинаково хорошее (т.е. точное) соответствие вызову. Однако версия с параметрами символьного массива более специализирована (см. раздел 16.3), она и выбирается для этого вызова.

Если бы была определена версия функции `compare()`, получающая указатели на символы, как простая, не шаблонная функция (а не как специализация шаблона), то этот вызов разрешится по-другому. В данном случае было бы три подходящих функции: эти два шаблона и не шаблонная версия указателя на символ. Все три одинаково хорошо подходят для этого вызова. Как уже упоминалось, когда нешаблонная функция обеспечивает одинаково хорошее соответствие с шаблонной, выбирается нешаблонная функция (см. раздел 16.3).

Ключевая концепция. Обычные правила области видимости относятся и к специализации

Чтобы специализировать шаблон, объявление его оригинала должно быть в области видимости. Кроме того, объявление специализации должно быть в области видимости перед любым кодом, использующим экземпляр шаблона.

Пропуск объявления обычных классов и функций найти очень просто — компилятор не сможет обработать такой код. Но при отсутствии

объявления специализации компилятор обычно создает код, используя первоначальный шаблон. Поэтому ошибки в порядке объявления шаблона и его специализации довольно просто допустить, но очень трудно найти.

Использование специализации и экземпляра первоначального шаблона с тем же набором аргументов шаблона является ошибкой. Но компилятор вряд ли обнаружит эту ошибку.



Рекомендуем

Шаблоны и их специализации должны быть объявлены в том же файле заголовка. Объявления всех шаблонов с данным именем должны располагаться сначала, а затем все специализации этих шаблонов.

Специализация шаблона класса

Кроме специализации шаблонов функций, вполне можно также специализировать шаблоны классов. В качестве примера определим специализацию библиотечного шаблона `hash`, который можно использовать для хранения объектов класса `Sales_data` в неупорядоченном контейнере. По умолчанию неупорядоченные контейнеры используют для организации своих элементов класс `hash<key_type>` (см. раздел 11.4). Чтобы использовать его с собственным типом данных, следует определить специализацию шаблона `hash`. Специализированный класс `hash` должен определять следующее.

- Перегруженный оператор вызова (см. раздел 14.8), возвращающий тип `size_t` и получающий объект типа ключа контейнера.
- Два члена-типа `result_type` и `argument_type`, соответствующие типу возвращаемого значения и типу аргумента оператора вызова.
- Стандартный конструктор и оператор присвоения копии, которые могут быть определены неявно (см. раздел 13.1.2).

Единственное осложнение в определении этой специализации класса `hash` состоит в том, что специализация шаблона должна быть в том же пространстве имен, в котором определяется первоначальный шаблон. Более подробная информация о пространствах имен приведена в разделе 18.2, а пока достаточно знать, что к пространству имен можно добавлять члены. Для этого следует сначала открыть пространство имен:

```
// открыть пространство имен std, чтобы можно было
```

специализировать

```
// класс std::hash
namespace std {
} // закрыть пространство имен std; обратите внимание: никакой точки с
```

// запятой после закрывающей фигурной скобки

Любые определения, расположенные между открывающей и закрывающей фигурными скобками, будут частью пространства имен std.

Следующий код определяет специализацию класса hash для класса Sales_data:

```
// открыть пространство имен std, чтобы можно было специализировать
// класс std::hash
namespace std {
template <> // определение специализации с параметром
struct hash<Sales_data> // шаблона класса Sales_data
{
    // тип, используемый для неупорядоченного контейнера hash, должен
    // определять следующие типы
    typedef size_t result_type;
    typedef Sales_data argument_type; // по умолчанию этому типу // требуется
                                    // оператор ==
    size_t operator()(const Sales_data& s) const;
    // класс использует синтезируемые функции управления копированием
    // и стандартный конструктор
};
size_t
hash<Sales_data>::operator()(const Sales_data& s) const {
    return hash<string>()(s.bookNo) ^
           hash<unsigned>()(s.units_sold) ^
           hash<double>()(s.revenue);
}
```

```
 } // закрыть пространство имен std; обратите  
внимание: никакой точки с  
 // запятой после закрывающей фигурной скобки
```

Определение `hash<Sales_data>` начинается с части `template<>`, означающей, что определяется полностью специализированный шаблон. Специализируемый шаблон называется `hash`, а специализированная версия — `hash<Sales_data>`. Члены класса следуют непосредственно из требований для специализации шаблона `hash`.

Подобно любым другим классам, специализируемые члены можно определить в классе или вне его, как это сделано здесь. Перегруженный оператор вызова должен определять хеш-функцию по значениям заданного типа. Эта функция обязана возвращать каждый раз тот же результат, когда она вызывается для данного значения. Хеш-функция практически всегда возвращает другой результат для не равных объектов.

Все сложности определения хорошей хеш-функции делегируем библиотеке. Библиотека определяет специализации класса `hash` для встроенных типов и для большинства библиотечных типов. Безымянный объект `hash<string>` используется для создания хеш-кода для переменной-члена `bookNo`, объект типа `hash<unsigned>` для создания хеш-кода из переменной-члена `units_sold` и объекта типа `hash<double>` для создания хеш-кода из переменной-члена `revenue`. Применение к этим результатам оператора исключающего ИЛИ (см. раздел 4.8) сформирует общий хеш-код для заданного объекта класса `Sales_data`.

Следует заметить, что хеш-функция определена для хеширования всех трех переменных-членов, чтобы она была совместима с определением оператора `operator==` класса `Sales_data` (см. раздел 14.3.1). По умолчанию неупорядоченные контейнеры используют специализацию хеша, соответствующую типу `key_type`, наряду с оператором равенства типа ключа.

С учетом того, что специализация находится в области видимости, она будет использоваться автоматически при использовании класса `Sales_data` как ключ в одном из этих контейнеров:

```
// использует hash<Sales_data> и оператор  
operator== класса Sales_data  
// из раздела 14.3.1  
unordered_multiset<Sales_data> SDset;  
Поскольку hash<Sales_data> использует закрытые члены класса
```

`Sales_data`, этот класс следует сделать другом класса `Sales_data`:

```
template <class T> class std::hash; // нужно для  
объявления
```

```
// дружественным  
class Sales_data {  
    friend class std::hash<Sales_data>;  
    // другие члены, как прежде  
};
```

Здесь указано, что специфический экземпляр `hash<Sales_data>` является дружественным. Поскольку данный экземпляр определяется в пространстве имен `std`, следует помнить, что этот тип хеша определяется в пространстве имен `std`. Следовательно, объявление `friend` относится к `std::hash`.



Чтобы позволить пользователям класса `Sales_data` использовать специализацию шаблона `hash`, следует определить эту специализацию в заголовке `Sales_data`.

Частичная специализация шаблона класса

В отличие от шаблона функции, специализация шаблона класса не обязана предоставлять аргументы для каждого параметра шаблона. Можно определить некоторые из них, но не все.

Частичная специализация (partial specialization) шаблона класса сама является шаблоном. Пользователи должны предоставить аргументы для тех параметров шаблона, которые не затронуты специализацией.



Частично можно специализировать только шаблон класса. Нельзя частично специализировать шаблон функции.

Библиотечный тип `remove_reference` был представлен в разделе 16.2.3, он работает с серией специализаций:

```

// первоначальный, наиболее общий шаблон
template <class T> struct remove_reference {
    typedef T type;
};

// частичные специализации, которые будут
использоваться для ссылок
// на l- и r-значения
template <class T> struct remove_reference<T&> // 
ссылки на l-значение
{ typedef T type; };
template <class T> struct remove_reference<T&&> // 
ссылки на r-значение
{ typedef T type; };

```

Первый шаблон определяет самую общую версию. Его экземпляр может быть создан с любым типом; он использует свой аргумент шаблона как тип для своего члена `type`. Следующие два класса — это частичные специализации первоначального шаблона.

Поскольку частичная специализация — это шаблон, начнем, как обычно, с определения параметров шаблона. Подобно любой другой специализации, у частичной специализации то же имя, что и у специализируемого шаблона. Список параметров специализации шаблона включает элементы для каждого параметра шаблона, тип которого не был определен полностью при частичной специализации. После имени класса располагаются аргументы для параметров специализируемого шаблона. Эти аргументы располагаются в угловых скобках после имени шаблона. Аргументы позиционально соответствуют параметрам первоначального шаблона.

Список параметров шаблона частичной специализации — это подмножество или специализация списка параметров первоначального шаблона. В данном случае у специализаций то же количество параметров, что и у первоначального шаблона. Но тип параметров в специализациях отличается от первоначального шаблона. Специализация будут использоваться для ссылок на типы l- и r-значений соответственно:

```

int i;
// decltype(42) - это int, используется
первоначальный шаблон
remove_reference<decltype(42)>::type a;
// decltype(i) - это int&, используется первая (T&)
частичная

```

```

// специализация
remove_reference<decltype(i)>::type b;
// decltype(std::move(i)) - это int&&, используется
вторая (т.е., T&&)
// частичная специализация
remove_reference<decltype(std::move(i))>::type c;
У всех трех переменных, a, b и c, тип int.

```

Специализация членов, но не класса

Вместо специализации всего шаблона можно специализировать только одну или несколько его функций-членов. Например, если Foo — это шаблон класса с членом Bar, можно специализировать только этот член:

```

template <typename T> struct Foo {
    Foo (const T &t = T()): mem(t) { }
    void Bar() { /* ... */ }
    T mem;
    // другие члены класса Foo
};

template<> // специализация шаблона
void Foo<int>::Bar() // специализация члена Bar
класса Foo<int>
{
    // осуществить всю специализированную обработку,
    // относящуюся к целым
    // числам
}

// осуществить всю специализированную обработку,
// относящуюся к целым
// числам

```

Здесь специализируется только один член класса Foo<int>. Другие его члены предоставляются шаблоном Foo:

```

Foo<string>    fs;      // создает экземпляр
Foo<string>::Foo()
    fs.Bar();           // создает экземпляр
Foo<string>::Bar()

Foo<int>    fi;        // создает экземпляр
Foo<int>::Foo()
    fi.Bar();          // использует специализацию
Foo<int>::Bar()

```

При использовании шаблона Foo с любым типом, кроме int, члены экземпляра создаются, как обычно. При использовании шаблона Foo с

типов `int` все члены экземпляра, кроме `Bar`, создаются, как обычно. Если использовать член `Bar` класса `Foo<int>`, то получится специализированное определение.

Упражнения раздела 16.5

Упражнение 16.62. Определите собственную версию класса `hash<Sales_data>` и контейнер `unordered_multiset` объектов класса `Sales_data`. Поместите в контейнер несколько транзакций и выведите его содержимое.

Упражнение 16.63. Определите шаблон функции для подсчета количества вхождений заданного значения в векторе. Проверьте программу, передав ей вектор значений типа `double`, вектор целых чисел и вектор строк.

Упражнение 16.64. Напишите специализированную версию шаблона из предыдущего упражнения для обработки вектора `vector<const char*>` и используйте ее в программе.

Упражнение 16.65. В разделе 16.3 были определены две перегруженных версии функции `debug_rep()`, одна из которых получает параметр типа `const char*`, а вторая — типа `char*`. Перепишите эти функции как специализации.

Упражнение 16.66. Каковы преимущества и недостатки перегрузки функций `debug_rep()` по сравнению с определением специализаций?

Упражнение 16.67. Повлияет ли определение этих специализаций на подбор функций `debug_rep()`? Почему?

Резюме

Шаблоны — это отличительная особенность языка C++ и основа его стандартной библиотеки. Шаблон представляет собой независимый от типа "чертеж", используемый компилятором для создания конкретных экземпляров указанных классов или функций. Шаблон разрабатывается один раз, а его экземпляры компилятор создает для соответствующего типа или значения по мере его применения.

Можно определять шаблоны функций и классов. Библиотечные алгоритмы являются шаблонами функций, а библиотечные контейнеры — шаблонами классов.

Явный аргумент шаблона позволяет фиксировать тип или значение одного или нескольких параметров шаблона. К параметрам с явным аргументом шаблона применимы нормальные преобразования.

Специализация шаблона — это отдельное специальное определение, позволяющее создать такую версию шаблона, в которой для одного или нескольких параметров указан определенный тип или значение. Специализация полезна в случае, когда для некоторых типов стандартное определение шаблона неприменимо.

Главная часть последнего выпуска стандарта языка C++ относится к шаблонам с переменным количеством аргументов. Такой шаблон способен получать переменное количество аргументов разных типов. Шаблоны с переменным количеством аргументов позволяют написать такие функции, как функция-член `emplace()` классов контейнеров и библиотечная функция `make_shared()`, передающая аргументы конструктору объекта.

Термины

Аргумент шаблона (template argument). Тип или значение, указанные при создании экземпляра шаблона.

Аргумент шаблона по умолчанию (default template argument). Тип или значение, используемые при создании экземпляра шаблона, если пользователь не предоставил соответствующий аргумент.

Дедукция аргумента шаблона (template argument deduction). Процесс, в ходе которого компилятор выясняет, какой экземпляр шаблона функции следует создать. Для этого компилятор исследует типы аргументов, переданных в качестве параметров шаблона. На основании полученных типов или значений объектов, связанных с параметрами шаблона, компилятор автоматически создает соответствующую версию функции.

Пакет параметров (parameter pack). Параметр шаблона или функции, представляющий любое количество параметров.

Пакет параметров функции (function parameter pack). Пакет, представляющий любое количество параметров функций.

Пакет параметров шаблона (template parameter pack). Пакет, представляющий любое количество параметров шаблона.

Параметр значения (nontype parameter). Параметр шаблона, представляющий значение. Во время создания экземпляра шаблона класса каждый параметр значения связывается с константным выражением, переданным в качестве аргумента при создании экземпляра класса.

Параметр типа (type parameter). Имя, используемое в списке параметров шаблона вместо имени типа. Параметры типа определяются после ключевого слова `typename` или `class`.

Параметр шаблона (template parameter). Имя, определенное в списке

параметров шаблона и используемое в определении его экземпляров. Параметр шаблона может быть типом или значением. Чтобы использовать шаблон класса, следует предоставить явные аргументы для каждого параметра шаблона. Компилятор использует эти типы или значения при создании версии экземпляра класса. При этом используемые параметры заменяются фактическими аргументами. Когда используется шаблон функции, компилятор выводит аргументы шаблона из аргументов вызова и создает экземпляр специфической функции на их основании.

Развертывание пакета (pack expansion). Процесс, в ходе которого пакет параметров заменяется соответствующим списком его элементов.

Создание экземпляра (instantiation). Процесс компилятора, в ходе которого соответствующие параметры шаблона заменяются фактическими аргументами и создается специфический экземпляр шаблона. Экземпляры функций создаются автоматически на основании аргументов, использованных в вызове. При использовании шаблона класса следует явно предоставить аргументы шаблона.

Создание экземпляра (instantiation). Процесс создания компилятором класса или функции из шаблона.

Специализация шаблона (template specialization). Переопределение всего шаблона класса, или его члена, или шаблона функции, в котором определены параметры шаблона. Специализация шаблона не может быть осуществлена до завершения определения шаблона класса, подвергающегося специализации. Специализация шаблона должна быть осуществлена прежде, чем он будет использован для специализированных аргументов. Каждый параметр шаблона в шаблоне функции должен быть специализирован полностью.

Список параметров шаблона (template parameter list). Список параметров типа или значения (разделяемый запятыми), используемый в определении или объявлении шаблона.

Схема (pattern). Определяет форму каждого элемента в развернутом пакете параметров.

Трансформация типа (type transformation). Определенные библиотекой шаблоны класса, преобразующие предоставленный параметр типа шаблона в связанный тип.

Частичная специализация (partial specialization). Версия шаблона класса, в которой определены некоторые, но не все параметры шаблона либо некоторые параметры определены не полностью.

Шаблон класса (class template). Определение, которое может быть использовано при создании экземпляров специфических классов. При

определении шаблона класса используется ключевое слово `template`, за которым следует разделяемый запятыми список параметров, заключенный в угловые скобки (`<>`).

Шаблон с переменным количеством аргументов (variadic template). Шаблон, получающий переменное количество аргументов. Пакет параметров шаблона определяется с использованием многоточия (например, `class... tурename...` или `имя_типа...`).

Шаблон функции (function template). Определение, которое может быть использовано при создании экземпляра специфической функции. При определении шаблона функции используется ключевое слово `template`, за которым следует разделяемый запятыми список параметров, заключенный в угловые скобки (`<>`), и определение функции.

Шаблон-член (member template). Член класса или шаблона класса, который является шаблоном функции. Шаблон-член не может быть виртуальным.

Явное создание экземпляра (explicit instantiation). Объявление, предоставляющее явные аргументы для всех параметров шаблона. Используется для управления процессом создания экземпляра. Если объявление будет внешним (`extern`), то экземпляр шаблона не будет создан; в противном случае создается экземпляр шаблона с указанными аргументами. Для каждого внешнего объявления шаблона где-нибудь в программе должно быть внутреннее явное создание экземпляра.

Явный аргумент шаблона (explicit template argument). Аргумент шаблона, предоставляемый пользователем при вызове функции или определении типа шаблона класса. Явные аргументы шаблона указывают в угловых скобках непосредственно после имени шаблона.

Часть IV

Дополнительные темы

Часть IV посвящена дополнительным средствам, которые весьма полезны в некоторых случаях, но нужны не каждому разработчику на языке C++. Эти средства делятся на две группы: те, которые используются для решения крупномасштабных проблем, и те, которые применяют скорее для специфических целей, а не общих. Средства для специфических задач, предоставляемые языком, рассматриваются в главе 19, а таковые, предоставленные библиотекой, — в главе 17.

В главе 17 рассматриваются четыре библиотечных средства специального назначения: класс `bitset` (набора битов) и три новых библиотечных средства: кортежи, регулярные выражения и случайные числа. Затронуты также будут и некоторые из менее общеизвестных частей библиотеки ввода и вывода.

Глава 18 посвящена обработке исключений, пространствам имен и множественному наследованию. Эти средства могут быть весьма полезны в контексте крупномасштабных программ.

Даже достаточно простые программы, которые могут быть написаны одним разработчиком, способны извлечь пользу из обработки исключений, основы которой были представлены в главе 5. Однако необходимость справляться с непредвиденными ошибками во время выполнения программы не менее важна, чем решение проблем в больших группах разработчиков. В главе 18 представлен обзор некоторых дополнительных средств обработки исключений. Здесь также более подробно рассматриваются способы обработки исключений, их смысл при размещении ресурсов в памяти и их удалении. Кроме того, в этой главе описаны способы создания и применения собственных классов исключений, рассматриваются также усовершенствования из нового стандарта, включая определение того, что некая функция не будет передавать исключения.

В крупномасштабных приложениях зачастую используют код от нескольких независимых производителей. Комбинирование нескольких библиотек от независимых разработчиков было бы необычайно трудной или вообще неразрешимой задачей, если бы все использованные в них имена располагались в одном пространстве имен. В библиотеках от независимых разработчиков почти неизбежно использовались бы

совпадающие имена. В результате имя, определенное в одной библиотеке, вступило бы в конфликт с таким же именем из другой библиотеки. Чтобы избежать конфликтов имен, их следует определять в *пространстве имен* (*namespace*).

Каждый раз, когда в этой книге использовалось имя из стандартной библиотеки, происходило обращение к пространству имен `std`. В главе 18 продемонстрировано, как можно определять собственные пространства имен.

Глава 18 завершается очень важным, но нечасто используемым средством языка: множественным наследованием. Множественное наследование наиболее полезно в сложных иерархиях наследования.

Глава 19 посвящена ряду специализированных подходов и инструментальных средств решения ряда специфических проблем. В этой главе рассматриваются такие средства, как дополнительные возможности по распределению памяти; поддержка языком C++ идентификации типов времени выполнения (RTTI), позволяющей определять фактический тип выражения во время выполнения; а также способы определения и использования указателей на члены класса. Указатели на члены классов отличаются от указателей на обычные данные или функции. Указатели на члены класса должны также отражать класс, которому принадлежит член. Затем рассматриваются три дополнительных составных типа: объединения, вложенные и локальные классы. Глава завершается кратким обзором средств, применение которых делает код непереносимым. Сюда относится спецификатор `volatile`, битовые поля и директивы компоновки.

Глава 17

Специализированные средства библиотек

Последний стандарт существенно увеличил размер и область видимости библиотеки. Действительно, посвященная библиотеке часть стандарта более чем удвоилась по сравнению с прежним выпуском стандарта и составила почти две трети текста нового стандарта. В результате подробное рассмотрение каждого класса библиотеки C++ стало невозможным в данном издании. Однако четыре специализированных библиотечных средства являются достаточно общими, чтобы рассмотреть их в данной книге: это кортежи, наборы битов, генераторы случайных чисел и регулярные выражения. Кроме того, будут рассмотрены также некоторые дополнительные специальные средства библиотеки ввода и вывода.

17.1. Тип `tuple`

C++
11

Шаблон `tuple` (кортеж) подобен шаблону `pair` (пара) (см. раздел 11.2.3). У каждого экземпляра шаблона `pair` могут быть члены разных типов, но их всегда только два. Члены экземпляров шаблона `tuple` также могут иметь разные типы, но количество их может быть любым. Каждый конкретный экземпляр шаблона `tuple` имеет фиксированное количество членов, но другой экземпляр типа может отличаться количеством членов.

Тип `tuple` особенно полезен, когда необходимо объединить некоторые данные в единый объект, но нет желания определять структуру для их хранения. Список операций, поддерживаемых типом `tuple`, приведен в табл. 17.1. Тип `tuple`, наряду с сопутствующими ему типами и функциями, определен в заголовке `tuple`.

Таблица 17.1. Операции с кортежами

<code>tuple<T1, T2, ..., Tn> t;</code>	<code>t</code> — кортеж с количеством и типами членов, заданными списком <code>T1... Tn</code> . Члены инициализируются по умолчанию (см. раздел 3.3.1)
<code>tuple<T1, T2, ..., Tn> t(v1, v2, ..., vn);</code>	<code>t</code> — кортеж с типами <code>T1... Tn</code> , каждый член которого инициализируется соответствующим инициализатором <code>vi</code> . Этот конструктор является явным (см. раздел 7.5.4)
<code>make_tuple(v1, v2, ..., vn)</code>	Возвращает кортеж, инициализированный данными инициализаторов. Тип кортежа выводится из типов инициализаторов
<code>t1 == t2</code> <code>t1 != t2</code>	Два кортежа равны, если у них совпадает количество членов и каждая пара членов равна. Для сравнения используется собственный оператор <code>==</code> каждого члена. Как только найдены неравные члены, последующие не проверяются
<code>t1 < t2</code>	Операторы сравнения кортежей используют алфавитный порядок (см. раздел 9.2.7). У кортежей должно быть одинаковое количество членов. Члены кортежа <code>t1</code> сравниваются с соответствующими членами кортежа <code>t2</code> при помощи оператора <code><</code>
	Возвращает ссылку <code>i</code> -ю переменную-член

<code>get<i>(t)</code>	кортежа <code>t</code> ; если <code>t</code> — это <code>l</code> -значение, то результат — ссылка на <code>l</code> -значение; в противном случае — ссылка на <code>r</code> -значение. Все члены кортежа являются открытыми (<code>public</code>)
<code>tuple_size<типКортежа>:: value</code>	Шаблон класса, экземпляр которого может быть создан по типу кортежа и имеет <code>public constexpr static</code> переменную-член <code>value</code> типа <code>size_t</code> , содержащую количество членов в указанном типе кортежа
<code>tuple_element<i, типКортежа>:: type</code>	Шаблон класса, экземпляр которого может быть создан по целочисленной константе и типу кортежа, имеющий открытый член <code>type</code> , являющийся типом указанного члена в кортеже указанного типа



Тип `tuple` можно считать структурой данных на "скорую руку".

17.1.1. Определение и инициализация кортежей

При определении кортежа следует указать типы каждого из его членов:

```
tuple<size_t, size_t, size_t> threeD; // все три
члена установлены в 0
tuple<string, vector<double>, int, list<int>>
    someVal("constants", { 3.14, 2.718}, 42,
{ 0, 1, 2, 3, 4, 5});
```

При создании объекта кортежа можно использовать либо стандартный конструктор кортежа, инициализирующий каждый член по умолчанию (см. раздел 3.3.1), либо предоставить инициализатор для каждого члена, как при инициализации кортежа `someVal`. Этот конструктор кортежа является явным (см. раздел 7.5.4), поэтому следует использовать прямой синтаксис инициализации:

```
tuple<size_t, size_t, size_t> threeD = { 1, 2, 3}; // ошибка
tuple<size_t, size_t, size_t> threeD{ 1, 2, 3}; // ok
```

В качестве альтернативы, подобно функции `make_pair()` (см. раздел 11.2.3), можно использовать библиотечную функцию `make_tuple()`, создающую объект кортежа:

```
// кортеж, представляющий транзакцию приложения  
книжного магазина:
```

```
// ISBN, количество, цена книги
```

```
auto item = make_tuple("0-999-78345-X", 3, 20.00);
```

Подобно функции `make_pair()`, функция `make_tuple()` использует типы, предоставляемые в качестве инициализаторов, для вывода типа кортежа. В данном случае кортеж `item` имеет тип `tuple<const char*, int, double>`.

Доступ к членам кортежа

В типе `pair` всегда есть два члена, что позволяет библиотеке присвоить им имена `first` (первый) и `second` (второй). Для типа `tuple` такое соглашение об именовании невозможно, поскольку у него нет ограничений на количество членов. В результате члены остаются безымянными. Вместо имен для обращения к членам кортежа используется библиотечный шаблон функции `get`. Чтобы использовать шаблон `get`, следует определить явный аргумент шаблона (см. раздел 16.2.2), задающий позицию члена, доступ к которому предстоит получить. Функция `get()` получает объект кортежа и возвращает ссылку на его заданный член:

```
auto book = get<0>(item);           // возвращает первый  
член item
```

```
auto cnt = get<1>(item);           // возвращает второй  
член item
```

```
auto price = get<2>(item) / cnt;    // возвращает  
последний член item
```

```
get<2>(item) *= 0.8;               // применяет 20%-ную  
скидку
```

Значение в скобках должно быть целочисленным константным выражением (см. разделе 2.4.4). Как обычно, счет начинается с 0, а значит, первым членом будет `get<0>`.

Если подробности типов в кортеже неизвестны, для выяснения количества и типов его членов можно использовать два вспомогательных шаблона класса:

```
typedef decltype(item) trans; // trans - тип  
кортежа item
```

```
// возвращает количество членов в объекте типа  
trans
```

```
size_t sz = tuple_size<trans>::value;
```

```

// возвращает 3
// cnt имеет тот же тип, что и второй член item
tuple_element<1, trans>::type cnt = get<1>(item);
// cnt - это int

```

Для использования шаблонов `tuple_size` и `tuple_element` необходимо знать тип объекта кортежа. Как обычно, проще всего определить тип объекта при помощи спецификатора `decltype` (см. раздел 2.5.3). Здесь спецификатор `decltype` используется для определения псевдонима для типа кортежа `item`, который и используется при создании экземпляров обоих шаблонов.

Шаблон `tuple_size` обладает открытой статической переменной-членом `value`, содержащей количество членов в указанном кортеже. Шаблон `tuple_element` получает индекс, а также тип кортежа. Он обладает открытым типом-членом `type`, содержащим тип указанного члена кортежа заданного типа. Подобно функции `get()`, шаблон `tuple_element` ведет отсчет индексов начиная с нуля.

Операторы сравнения и равенства

Операторы сравнения и равенства кортежей ведут себя подобно соответствующим операторам контейнеров (см. раздел 9.2.7). Эти операторы выполняются для членов двух кортежей, слева и справа. Сравнить два кортежа можно только при совпадении количества их членов. Кроме того, чтобы использовать операторы равенства или неравенства, должно быть допустимо сравнение каждой пары членов при помощи оператора `==`; а для использования операторов сравнения допустимым должно быть использование оператора `<`. Например:

```

tuple<string, string> duo( "1", "2" );
tuple<size_t, size_t> twoD( 1, 2 );
bool b = (duo == twoD); // ошибка: нельзя сравнить
                        // size_t и string
tuple<size_t, size_t, size_t> threeD( 1, 2, 3 );
b = (twoD < threeD);           // ошибка: разное
                                // количество членов
tuple<size_t, size_t> origin( 0, 0 );
b = (origin < twoD);          // ok: b - это true

```



Поскольку кортеж определяет операторы `<` и `==`, последовательности кортежей можно передавать алгоритмам, а также использовать кортеж как тип ключа в упорядоченном контейнере.

Упражнения раздела 17.1.1

Упражнение 17.1. Определите кортеж, содержащий три члена типа `int`, и инициализируйте их значениями 10, 20 и 30.

Упражнение 17.2. Определите кортеж, содержащий строку, вектор строки и пару из строки и целого числа (типы `string`, `vector<string>` и `pair<string, int>`).

Упражнение 17.3. Перепишите программы `TextQuery` из раздела 12.3 так, чтобы использовать кортеж вместо класса `QueryResult`. Объясните, что на ваш взгляд лучше и почему.

17.1.2. Использование кортежей для возвращения нескольких значений

Обычно кортеж используют для возвращения из функции нескольких значений. Например, рассматриваемый книжный магазин мог бы быть одним из нескольких магазинов в сети. У каждого магазина был бы транзакционный файл, содержащий данные по каждой проданной книге. В этом случае могло бы понадобиться просмотреть все продажи данной книги по всем магазинам.

Предположим, для каждого магазина имеется файл транзакций. Каждый из этих транзакционных файлов в магазине будет содержать все транзакции для каждой группы книг. Предположим также, что некая другая функция читает эти транзакционные файлы, создает вектор `vector<Sales_data>` для каждого магазина и помещает эти векторы в вектор векторов:

```
// каждый элемент в файле содержит транзакции  
// для определенного магазина  
vector<vector<Sales_data>> files;
```

Давайте напишем функцию, которая будет просматривать файлы в поисках магазина, продавшего заданную книгу. Для каждого магазина, у

которого есть соответствующая транзакция, необходимо создать кортеж для содержания индекса этого магазина и двух итераторов. Индекс будет позицией соответствующего магазина в файлах, а итераторы отметят первую и следующую после последней записи по заданной книге в векторе `vector<Sales_data>` этого магазина.

Функция, возвращающая кортеж

Для начала напишем функции поиска заданной книги. Аргументами этой функции будет только что описанный вектор векторов и строка, представляющая ISBN книги. Функция будет возвращать вектор кортежей с записями по каждому магазину, где была продана по крайней мере одна заданная книга:

```
// matches имеет три члена: индекс магазина и
итераторы в его векторе
typedef tuple<vector<Sales_data>::size_type,
               vector<Sales_data>::const_iterator,
               vector<Sales_data>::const_iterator>
matches;
// files хранит транзакции по каждому магазину
// findBook() возвращает вектор с записями для
каждого магазина,
// продавшего данную книгу
vector<matches>
findBook(const vector<vector<Sales_data>> &files,
         const string &book) {
    vector<matches> ret; // изначально пуст
    // для каждого магазина найти диапазон,
соответствующий книге
    // (если он есть)
    for (auto it = files.cbegin(); it != files.cend();
++it) {
        // найти диапазон Sales_data с тем же ISBN
        auto found = equal_range(it->cbegin(), it-
>cend(),
                                book, compareIsbn);
        if (found.first != found.second) // у этого
магазина есть продажи
            // запомнить индекс этого магазина и диапазона
соответствий
```

```

        ret.push_back( make_tuple( it - files.cbegin(),
                                    found.first,
                                    found.second) );
    }
    return ret; // пуст, если соответствий не найдено
}

```

Цикл `for` перебирает элементы вектора `files`, которые сами являются векторами. В цикле `for` происходит вызов библиотечного алгоритма `equal_range()`, работающего как одноименная функция-член ассоциативного контейнера (см. раздел 11.3.5). Первые два аргумента функции `equal_range()` являются итераторами, обозначающими исходную последовательность (см. раздел 10.1). Третий аргумент — значение. По умолчанию для сравнения элементов функция `equal_range()` использует оператор `<`. Поскольку тип `Sales_data` не имеет оператора `<`, передаем указатель на функцию `compareIsbn()` (см. раздел 11.2.2).

Алгоритм `equal_range()` возвращает пару итераторов, обозначающих диапазон элементов. Если книга не будет найдена, то итераторы окажутся равны, означая, что диапазон пуст. В противном случае первый член возвращенной пары обозначит первую соответствующую транзакцию, а второй — следующую после последней.

Использование возвращенного функцией кортежа

После создания вектора магазинов с соответствующей транзакцией эти транзакции необходимо обработать. В данной программе следует сообщить результаты общего объема продаж для каждого магазина, у которого была такая продажа:

```

void reportResults(istream &in, ostream &os,
                    const vector<vector<Sales_data>>
&files) {
    string s; // искомая книга
    while (in >> s) {
        auto trans = findBook(files, s);
        // магазин, продавший эту книгу
        if (trans.empty()) {
            cout << s << " not found in any stores" << endl;
            continue; // получить следующую книгу для поиска
        }
    }
}

```

```

        for (const auto &store : trans) // для каждого
магазина с
                                // продажей
    // get<n> возвращает указанный элемент кортежа в
store
    os << "store " << get<0>( store ) << " sales: "
    << accumulate( get<1>( store ), get<2>( store ),
    Sales_data( s ) )
    << endl;
}
}

```

Цикл `while` последовательно читает поток `istream` по имени `in`, чтобы запустить обработку следующей книги. Вызов функции `findBook()` позволяет выяснить, присутствует ли строка `s`, и присваивает результаты вектору `trans`. Чтобы упростить написание типа `trans`, являющегося вектором кортежей, используем ключевое слово `auto`.

Если вектор `trans` пуст, значит, по книге `s` никаких продаж не было. В таком случае выводится сообщение и происходит возврат к циклу `while`, чтобы обработать следующую книгу.

Цикл `for` свяжет ссылку `store` с каждым элементом вектора `trans`. Поскольку изменять элементы вектора `trans` не нужно, объявим ссылку `store` ссылкой на константу. Для вывода результатов используем `get`: `get<0>` — индекс соответствующего магазина; `get<1>` — итератор на первую транзакцию; `get<2>` — на следующую после последней.

Поскольку класс `Sales_data` определяет оператор суммы (см. раздел 14.3), для суммирования транзакций можно использовать библиотечный алгоритм `accumulate()` (см. раздел 10.2.1). Как отправную точку суммирования используем объект класса `Sales_data`, инициализированный конструктором `Sales_data()`, получающим строку (см. раздел 7.1.4). Этот конструктор инициализирует переменную-член `bookNo` переданной строкой, а переменные-члены `units_sold` и `revenue` — нулем.

Упражнения раздела 17.1.2

Упражнение 17.4. Напишите и проверьте собственную версию функции `findBook()`.

Упражнение 17.5. Перепишите функцию `findBook()` так, чтобы она

возвращала пару, содержащую индекс и пару итераторов.

Упражнение 17.6. Перепишите функцию `findBook()` так, чтобы она не использовала кортеж или пару.

Упражнение 17.7. Объясните, какую версию функции `findBook()` вы предпочитаете и почему.

Упражнение 17.8. Что будет, если в качестве третьего параметра алгоритма `accumulate()` в последнем примере кода этого раздела передать объект класса `Sales_data`?

17.2. Тип `bitset`

В разделе 4.8 приводились встроенные операторы, рассматривающие целочисленный operand как коллекцию битов. Для облегчения использования битовых операций и обеспечения возможности работы с коллекциями битов, размер которых больше самого длинного целочисленного типа, стандартная библиотека определяет класс `bitset` (набор битов). Класс `bitset` определен в заголовке `bitset`.

17.2.1. Определение и инициализация наборов битов

Список конструкторов типа `bitset` приведен в табл. 17.2. Тип `bitset` — это шаблон класса, который, подобно классу `array`, имеет фиксированный размер (см. раздел 3.3.6). При определении набора битов следует указать в угловых скобках количество битов, которые он будет содержать:

```
bitset<32> bitvec(1U); // 32 бита; младший бит 1,  
остальные биты 0
```

Размер должен быть указан константным выражением (см. раздел 2.4.4). Этот оператор определяет набор битов `bitvec`, содержащий 32 бита. Подобно элементам вектора, биты в наборе битов не имеют имен. Доступ к ним осуществляется по позиции. Нумерация битов начинается с 0. Таким образом, биты набора `bitvec` пронумерованы от 0 до 31. Биты, расположенные ближе к началу (к 0), называются *младшими битами* (low-order), а ближе к концу (к 31) — *старшими битами* (high-order).

Таблица 17.2. Способы инициализации набора битов

<code>bitset<n> b;</code>	Набор <code>b</code> содержит <code>n</code> битов, каждый из которых содержит значение 0. Это конструктор <code>constexpr</code> (см. раздел 7.5.6)
<code>bitset<n> b(u);</code>	Набор <code>b</code> содержит копию <code>n</code> младших битов значения <code>u</code> типа <code>unsigned long long</code> . Если значение <code>n</code> больше размера типа <code>unsigned long long</code> , остальные старшие биты устанавливаются на нуль. Это конструктор <code>constexpr</code> (см. раздел 7.5.6)
<code>bitset<n> b(s, pos, m, zero, one);</code>	Набор <code>b</code> содержит копию <code>m</code> символов из строки <code>s</code> , начиная с позиции <code>pos</code> . Стока <code>s</code> может содержать только символы для нулей и единиц; если строка <code>s</code> содержит любой другой символ, передается исключение <code>invalid_argument</code> . Символы хранятся в наборе <code>b</code> как нули и единицы соответственно. По умолчанию параметр <code>pos</code> имеет значение 0, параметр <code>m</code> — <code>string::npos</code> , <code>zero</code> — '0' и <code>one</code> — '1'

<code>bitset<n> b(cp, pos, m, zero, one);</code>	Подобен предыдущему конструктору, но копируется символьный массив, на который указывает <code>cp</code> . Если значение <code>m</code> не предоставлено, <code>cp</code> должен указывать на строку в стиле С. Если <code>m</code> предоставлено, то начиная с позиции <code>cp</code> в массиве должно быть по крайней мере <code>m</code> символов, соответствующих нулям или единицам
---	--

Конструкторы, получающие строку или символьный указатель, являются явными (см. раздел 7.5.4). В новом стандарте была добавлена возможность определять альтернативные символы для 0 и 1.

Инициализация набора битов беззнаковым значением

При использовании для инициализации набора битов целочисленного значения оно преобразуется в тип `unsigned long long` и рассматривается как битовая схема. Биты в наборе битов являются копией этой схемы. Если размер набора битов превосходит количество битов в типе `unsigned long long`, то остальные старшие биты устанавливаются в нуль. Если размер набора битов меньше количества битов, то будут использованы только младшие биты предоставленного значения, а старшие биты вне размера объекта набора битов отбрасываются:

```
// bitvec1 меньше инициализатора; старшие биты
инициализатора
// отбрасываются
bitset<13> bitvec1(0xbeef); // биты 1111011101111
// bitvec2 больше инициализатора; старшие биты
bitvec2
// устанавливаются в нуль
bitset<20> bitvec2(0xbeef); // биты
0000101111011101111
// на машинах с 64-битовым long long, 0ULL - это 64
бита из 0,
// a ~0ULL - 64 единицы
bitset<128> bitvec3(~0ULL); // биты 0...63 -
единицы; 63...121 - нули
```

Инициализация набора битов из строки

Набор битов можно инициализировать из строки или указателя на элемент в символьном массиве. В любом случае символы непосредственно представляют битовую схему. Как обычно, при использовании строки для представления числа символы с самыми низкими индексами в строке соответствуют старшим битам, и наоборот:

```
bitset<32> bitvec4( "1100" ); // биты 2 и 3 — единицы, остальные — 0
```

Если строка содержит меньше символов, чем битов в наборе, старшие биты устанавливаются в нуль.



Соглашения по индексации строк и наборов битов прямо противоположны: символ строки с самым высоким индексом (крайний правый символ) используется для инициализации младшего бита в наборе битов (бит с индексом 0). При инициализации набора битов из строки следует помнить об этом различии.

Необязательно использовать всю строку в качестве исходного значения для набора битов, вполне можно использовать часть строки:

```
string str( "1111111000000011001101" );
bitset<32> bitvec5( str, 5, 4 ); // четыре бита,
начиная с str[ 5 ] — 1100
bitset<32> bitvec6( str, str.size( ) - 4 ); // использует четыре последних
                                                // символа
```

Здесь набор битов `bitvec5` инициализируется подстрокой `str`, начиная с символа `str[5]`, и четырьмя символами далее. Как обычно, крайний справа символ подстроки представляет бит самого низкого порядка. Таким образом, набор `bitvec5` инициализируется битами с позиций 3 до 0 и получает значение 1100, а остальные биты — 0. Инициализатор набора битов `bitvec6` передает строку и отправную точку, поэтому он инициализируется символами строки `str`, начиная с четвертого и до конца строки `str`. Остаток битов набора `bitvec6` инициализируется нулями. Эти инициализации можно представить так:



Упражнения раздела 17.2.1

Упражнение 17.9. Объясните битовую схему, которую содержит каждый из следующих объектов `bitset`:

- (a) `bitset<64> bitvec(32);`
- (b) `bitset<32> bv(1010101);`
- (c) `string bstr; cin >> bstr; bitset<8> bv(bstr);`

17.2.2. Операции с наборами битов

Операции с наборами битов (табл. 17.3) определяют различные способы проверки и установки одного или нескольких битов. Класс `bitset` поддерживает также побитовые операторы, которые рассматривались в разделе 4.8. Применительно к объектам `bitset` эти операторы имеют тот же смысл, что и таковые встроенные операторы для типа `unsigned`.

Таблица 17.3. Операции с наборами битов

<code>b. any()</code>	Установлен ли в наборе <code>b</code> хоть какой-нибудь бит?
<code>b. all()</code>	Все ли биты набора <code>b</code> установлены?
<code>b. none()</code>	Нет ли в наборе <code>b</code> установленных битов?
<code>b. count()</code>	Количество установленных битов в наборе <code>b</code>
<code>b. size()</code>	Функция <code>constexpr</code> (см. раздел 2.4.4), возвращающая количество битов набора <code>b</code>
	Возвращает значение <code>true</code> , если бит в позиции <code>pos</code> установлен,

b. test(pos)	и значение <code>false</code> в противном случае
b. set(pos, v) b. set()	Устанавливает для бита в позиции <code>pos</code> логическое значение <code>v</code> . По умолчанию <code>v</code> имеет значение <code>true</code> . Без аргументов устанавливает все биты набора <code>b</code>
b. reset(pos) b. reset()	Сбрасывает бит в позиции <code>pos</code> или все биты набора <code>b</code>
b. flip(pos) b. flip()	Изменяет состояние бита в позиции <code>pos</code> или все биты набора <code>b</code>
b[pos]	Предоставляет доступ к биту набора <code>b</code> в позиции <code>pos</code> ; если набор <code>b</code> константен и бит установлен, то <code>b[pos]</code> возвращает логическое значение <code>true</code> , а в противном случае — значение <code>false</code>
b. to_ulong() b. to_ullong()	Возвращает значение типа <code>unsigned long</code> или типа <code>unsigned long long</code> с теми же битами, что и в наборе <code>b</code> . Если битовая схема в наборе <code>b</code> не соответствует указанному типу результата, передается исключение <code>overflow_error</code>
b. to_string(zero, one)	Возвращает строку, представляющую битовую схему набора <code>b</code> . Параметры <code>zero</code> и <code>one</code> имеют по умолчанию значения '0' и '1'. Они используются для представления битов 0 и 1 в наборе <code>b</code>
os << b	Выводит в поток <code>os</code> биты набора <code>b</code> как символы '0' и '1'
is >> b	Читает символы из потока <code>is</code> в набор <code>b</code> . Чтение прекращается, когда следующий символ отличается от 1 или 0 либо когда прочитано <code>b.size()</code> битов

Некоторые из функций, `count()`, `size()`, `all()`, `any()` и `none()`, не получают аргументов и возвращают информацию о состоянии всего набора битов. Другие, `set()`, `reset()` и `flip()`, изменяют состояние набора битов. Функции-члены, изменяющие набор битов, допускают перегрузку. В любом случае версия функции без аргументов применяется соответствующую операцию ко всему набору, а версии функций, получающих позицию, применяют операцию к заданному биту:

```
bitset<32> bitvec(1U); // 32 бита; младший бит 1,
остальные биты - 0
bool is_set = bitvec.any(); // true,
установлен один бит
bool is_not_set = bitvec.none(); // false,
установлен один бит
bool all_set = bitvec.all(); // false, только
один бит установлен
size_t onBits = bitvec.count(); // возвращает 1
```

```

size_t sz = bitvec.size();           // возвращает 32
bitvec.flip(); // инвертирует значения всех битов
в bitvec
bitvec.reset(); // сбрасывает все биты в 0
bitvec.set(); // устанавливает все биты в 1

```



Функция `any()` возвращает значение `true`, если один или несколько битов объекта класса `bitset` установлены, т.е. равны 1. Функция `none()`, наоборот, возвращает значение `true`, если все биты содержат нуль. Новый стандарт ввел функцию `all()`, возвращающую значение `true`, если все биты установлены. Функции `count()` и `size()` возвращают значение типа `size_t` (см. раздел 3.5.2), равное количеству установленных битов, или общее количество битов в объекте соответственно. Функция `size()` — `constexpr`, а значит, она применима там, где требуется константное выражение (см. раздел 2.4.4).

Функции `flip()`, `set()`, `reset()` и `test()` позволяют читать и записывать биты в заданную позицию:

```

bitvec.flip(0); // инвертирует значение первого
бита
bitvec.set(bitvec.size() - 1); // устанавливает
последний бит
bitvec.set(0, 0); // сбрасывает первый бит
bitvec.reset(i); // сбрасывает i-й бит
bitvec.test(0); // возвращает false, поскольку
первый бит сброшен

```

Оператор индексирования перегружается как константный. Константная версия возвращает логическое значение `true`, если бит по заданному индексу установлен, и значение `false` в противном случае. Неконстантная версия возвращает специальный тип, определенный классом `bitset`, позволяющий манипулировать битовым значением в позиции, заданной индексом:

```

bitvec[0] = 0; // сбрасывает бит в позиции
0
bitvec[31] = bitvec[0]; // устанавливает последний
бит в то же
// состояние, что и первый
bitvec[0].flip(); // инвертирует значение

```

```

бита в позиции 0
~bitvec[ 0]; // эквивалентная операция;
инвертирует бит
// в позиции 0
bool b = bitvec[ 0]; // преобразует значение
bitvec[ 0] в тип bool

```

Возвращение значений из набора битов

Функции `to_ulong()` и `to_ullong()` возвращают значение, содержащее ту же битовую схему, что и объект класса `bitset`. Эти функции можно использовать, только если размер набора битов меньше или равен размеру типа `unsigned long` для функции `to_ulong()` и типа `unsigned long long` для функции `to_ullong()` соответственно:

```

unsigned long ulong = bitvec3.to_ulong();
cout << "ulong = " << ulong << endl;

```



Если значение в наборе битов не соответствует заданному типу, эти функции передают исключение `overflow_error` (см. раздел 5.6).

Операторы ввода-вывода типа `bitset`

Оператор ввода читает символы из входного потока во временный объект типа `string`. Чтение продолжается, пока не будет заполнен соответствующий набор битов, или пока не встретится символ, отличный от 1 или 0, или не встретится конец файла, или ошибка ввода. Затем этой временной строкой (см. раздел 17.2.1) инициализируется набор битов. Если прочитано меньше символов, чем насчитывает набор битов, старшие биты, как обычно, устанавливаются в 0.

Оператор вывода выводит битовую схему объекта `bitset`:

```

bitset<16> bits;
cin >> bits; // читать до 16 символов 1 или 0 из
cin
cout << "bits: " << bits << endl; // вывести
прочитанное

```

Использование наборов битов

Для иллюстрации применения наборов битов повторно реализуем код оценки из раздела 4.8, использовавший тип `unsigned long` для представления результатов контрольных вопросов (сдал/не сдал) для 30 учеников:

```
bool status;
// версия, использующая побитовые операторы
unsigned long quizA = 0; // это значение
используется                                // как коллекция битов
quizA |= 1UL << 27;                      // отметить ученика номер
27 как сдавшего
status = quizA & (1UL << 27); // проверить оценку
ученика номер 27
quizA &= ~(1UL << 27);                  // ученик номер 27 не
сдал
// эквивалентные действия с использованием набора
битов
bitset<30> quizB;           // зарезервировать по одному
биту на студента; все
                                // биты инициализированы 0
quizB.set(27);                // отметить ученика номер 27
как сдавшего
status = quizB[27];          // проверить оценку ученика
номер 27
quizB.reset(27);              // ученик номер 27 не сдал
```

Упражнения раздела 17.2.2

Упражнение 17.10. Используя последовательность 1, 2, 3, 5, 8, 13, 21, инициализируйте набор битов, у которого установлена 1 в каждой позиции, соответствующей числу в этой последовательности. Инициализируйте по умолчанию другой набор битов и напишите небольшую программу для установки каждого из соответствующих битов.

Упражнение 17.11. Определите структуру данных, которая содержит целочисленный объект, позволяющий отследить (сдал/не сдал) ответы на контрольную из 10 вопросов. Какие изменения (если они вообще понадобятся) необходимо внести в структуру данных, если в контрольной станет 100 вопросов?

Упражнение 17.12. Используя структуру данных из предыдущего вопроса, напишите функцию, получающую номер вопроса и значение,

означающее правильный/неправильный ответ, и изменяющую результаты контрольной соответственно.

Упражнение 17.13. Создайте целочисленный объект, содержащий правильные ответы (да/нет) на вопросы контрольной. Используйте его для создания оценок контрольных вопросов для структуры данных из предыдущих двух упражнений.

17.3. Регулярные выражения

Регулярное выражение (regular expression) — это способ описания последовательности символов. Это чрезвычайно мощное средство программирования. Однако описание языков, используемых для определения регулярных выражений, выходит за рамки этой книги. Лучше сосредоточиться на использовании библиотеки регулярных выражений языка C++ (библиотеки RE), являющейся частью новой библиотеки. Библиотека RE определена в заголовке `regex` и задействует несколько компонентов, перечисленных в табл. 17.4.

Таблица 17.4. Компоненты библиотеки регулярных выражений

<code>regex</code>	Класс, представляющий регулярное выражение
<code>regex_match()</code>	Сравнивает последовательность символов с регулярным выражением
<code>regex_search()</code>	Находит первую последовательность, соответствующую регулярному выражению
<code>regex_replace()</code>	Заменяет регулярное выражение, используя заданный формат
<code>sregex_iterator</code>	Адаптер итератора, вызывающий функцию <code>regex_search()</code> для перебора совпадений в строке
<code>smatch</code>	Класс контейнера, содержащего результаты поиска в строке
<code>ssub_match</code>	Результаты совпадения выражений в строке



Если вы еще не знакомы с использованием регулярных выражений, то имеет смысл просмотреть этот раздел и выяснить, на что способны регулярные выражения.

Класс `regex` представляет регулярное выражение. Кроме инициализации и присвоения, с классом `regex` допустимо немного операций. Они перечислены в табл. 17.6.

Функции `regex_match()` и `regex_search()` определяют, соответствует ли заданная последовательность символов предоставленному объекту класса `regex`. Функция `regex_match()` возвращает значение `true`, если вся исходная последовательность

соответствует выражению; функция `regex_search()` возвращает значение `true`, если в исходной последовательности выражению соответствует подстрока. Есть также функция `regex_replace()`, описываемая в разделе 17.3.4.

Аргументы функции `regex` описаны в табл. 17.5. Эти функции возвращают логическое значение и допускают перегрузку: одна версия получает дополнительный аргумент типа `smatch`. Если он есть, эти функции сохраняют дополнительную информацию об успехе обнаружения соответствия в предоставленном объекте класса `smatch`.

17.3.1. Использование библиотеки регулярных выражений

В качестве довольно простого примера рассмотрим поиск слов, нарушающих известное правило правописания "i перед e, кроме как после c":

```
// найти символы ei, следующие за любым символом,
// кроме с
string pattern("[ ^c] ei");
// исходная схема должна присутствовать в целом
// слове
pattern = "[[:alpha:]]*" + pattern + "[[:alpha:]]*";
regex r(pattern); // создать regex для поиска схемы
smatch results; // определить объект для содержания
// результатов поиска
// определить строку, содержащую текст,
// соответствующий и не
// соответствующий схеме
string test_str = "receipt freind theif receive";
// использовать r для поиска соответствия в
test_str
if (regex_search(test_str, results, r)) // если
// соответствие есть
    cout << results.str() << endl; // вывести
// соответствующее слово
```

Таблица 17.5. Аргументы функций `regex_search()` и `regex_match()`

Обратите внимание: функции возвращают логическое значение, означающее, было ли найдено соответствие.

(seq, m, r, mft)	Поиск регулярного выражения объекта <code>r</code> класса <code>regex</code> в символьной последовательности <code>seq</code> . Последовательность <code>seq</code> может быть строкой, парой итераторов, обозначающих диапазон, или указателем на символьный массив с нулевым символом в конце, <code>m</code> — это объект соответствия, используемый для хранения подробностей о соответствии. Типы объекта <code>m</code> и последовательности <code>seq</code> должны быть совместимы (см. раздел 17.3.1). <code>mft</code> — это необязательное значение
------------------	--

`regex_constants::match_flag_type`. Это значение, описанное в табл. 17.13, влияет на процесс поиска соответствия

Таблица 17.6. Операции с классом `regex` (и `wregex`)

<code>regex r(re)</code> <code>regex r(re, f)</code>	Параметр <code>re</code> представляет регулярное выражение и может быть строкой, парой итераторов, обозначающих диапазон символов, указателем на символьный массив с нулевым символом в конце, указателем на символ и количеством или списком символов в скобках, <code>f</code> — это флаги, определяющие выполнение объекта. Флаги <code>f</code> устанавливаются исходя из упомянутых ниже значений. Если флаги <code>f</code> не определены, по умолчанию применяется ECMAScript
<code>r1 = re</code>	Заменяет регулярное выражение в <code>r1</code> регулярным выражением <code>re</code> . <code>re</code> — это регулярное выражение, которое может быть другим объектом класса <code>regex</code> , строкой, указателем на символьный массив с нулевым символом в конце или списком символов в скобках
<code>r1.assign(re,</code> <code>f)</code>	То же самое, что и оператор присвоения (<code>=</code>). Параметр <code>re</code> и необязательный флаг <code>f</code> имеют тот же смысл, что и соответствующие аргументы конструктора <code>regex()</code>
<code>r.mark_count()</code>	Количество подвыражений (рассматриваются в разделе 17.3.3) в объекте <code>r</code>
<code>r.flags()</code>	Возвращает набор флагов для объекта <code>r</code>

Примечание: конструкторы и операторы присвоения могут передавать исключение типа `regex_error`.

Флаги, применяемые при определении объекта класса `regex`. Определены в типах `regex` и `regex_constants::syntax_option_type`

<code>icase</code>	Игнорировать регистр при поиске соответствия
<code>nosubs</code>	Не хранить соответствия подвыражений
<code>optimize</code>	Предпочтение скорости выполнения скорости создания
<code>ECMAScript</code>	Использование грамматики согласно ECMA-262
<code>basic</code>	Использование базовой грамматики регулярных выражений POSIX
<code>extended</code>	Использование расширенной грамматики регулярных выражений POSIX
<code>awk</code>	Использование грамматики POSIX версии языка awk
<code>grep</code>	Использование грамматики POSIX версии языка grep
<code>egrep</code>	Использование грамматики POSIX версии языка egrep

Начнем с определения строки для хранения искомого регулярного выражения. Регулярное выражение `[^c]` означает любой символ, отличный от символа '`c`', а `[^c] ei` — любой такой символ,

сопровождаемый символами 'ei'. Эта схема описывает строки, содержащие только три символа. Необходимо найти целое слово, содержащее эту схему. Для соответствия слову необходимо регулярное выражение, которое будет соответствовать символам, расположенным прежде и после заданной трехсимвольной схемы.

Это регулярное выражение состоит из любого количества символов, сопровождаемых первоначальной трехсимвольной схемой и любым количеством дополнительных символов. По умолчанию объекты класса `regex` используют язык регулярных выражений ECMAScript. На языке ECMAScript схема `[[:alpha:]]` соответствует любому алфавитному символу, а символы `+` и `*` означают "один или несколько" и "нуль или более" соответственно. Таким образом, схема `[[:alpha:]]*` будет соответствовать любому количеству символов.

Регулярное выражение, сохраненное в строке `pattern`, используется для инициализации объекта `r` класса `regex`. Затем определяется строка, которая будет использована для проверки регулярного выражения. Стока `test_str` инициализируется словами, которые соответствуют схеме (например, "freind" и "theif"), и словами, которые ей не соответствуют (например, "receipt" и "receive"). Определим также объект `results` класса `smatch`, передаваемый функции `regex_search()`. Если соответствие будет найдено, то объект `results` будет содержать подробности о том, где оно найдено.

Затем происходит вызов функции `regex_search()`. Если она находит соответствие, то возвращает значение `true`. Для вывода части строки `test_str`, соответствующей заданной схеме, используется функция-член `str()` объекта `results`. Функция `regex_search()` прекращает поиск, как только находит в исходной последовательности соответствующую подстроку. В результате вывод будет таким:

```
freind
```

Поиск всех соответствий во вводе представлен в разделе 17.3.2.

Определение параметров объекта regex

При определении объекта класса `regex` или вызове его функции `assign()` для присвоения ему нового значения можно применить один или несколько флагов, влияющих на работу объекта класса `regex`. Эти флаги контролируют обработку, осуществляющую этим объектом. Последние шесть флагов, указанных в табл. 17.6, задают язык, на котором написано регулярное выражение. Установлен должен быть только один из

флагов определения языка. По умолчанию установлен флаг ECMAScript, задающий использование объектом класса `regex` спецификации ECMA-262, являющейся языком регулярных выражений большинства веб-браузеров.

Другие три флага позволяют определять независимые от языка аспекты обработки регулярного выражения. Например, можно указать, что поиск регулярного выражения не будет зависеть от регистра символов.

В качестве примера используем флаг `icase` для поиска имен файлов с указанными расширениями. Большинство операционных систем распознают расширения без учета регистра символов: программа C++ может быть сохранена в файле с расширением `.cc`, `.Cc`, `.cC` или `.CC`. Давайте напишем регулярное выражение для распознавания любого из них наряду с другими общепринятыми расширениями файлов:

```
// один или несколько алфавитно-цифровые символы,
// сопровождаемых
// и "cpp", "cxx" или "cc"
regex r("[[:alnum:]]+\.\(cpp|cxx|cc)\$",
regex::icase);
smatch results;
string filename;
while (cin >> filename)
    if (regex_search(filename, results, r))
        cout << results.str() << endl; // вывод текущего
соответствия
```

Это выражение будет соответствовать строке из одного или нескольких символов или цифр, сопровождаемых точкой и одним из трех расширений файла. Регулярное выражение будет соответствовать расширению файлов независимо от регистра.

Подобно тому, как специальные символы есть в языке C++ (см. раздел 2.1.3), у языков регулярных выражений, как правило, тоже есть специальные символы. Например, точка `(.)` обычно соответствует любому символу. Как и в языке C++, для обозначения специального характера символа его предваряют символом наклонной черты. Поскольку наклонная черта влево является также специальным символом в языке C++, в строковом литерале языка C++, означающем наклонную черту влево следует использовать вторую наклонную черту влево. Следовательно, чтобы представить точку в регулярном выражении, необходимо написать `\.\.`.

Ошибки в определении и использовании регулярного выражения

Регулярное выражение можно считать самостоятельной "программой" на простом языке программирования. Этот язык не интерпретируется компилятором C++, и "компилируется" только во время выполнения, когда объект класса `regex` инициализируется или присваивается. Как и в любой написанной программе, в регулярных выражениях вполне возможны ошибки.



Важно понимать, что правильность синтаксиса регулярного выражения проверяется во время выполнения.

Если допустить ошибку в записи регулярного выражения, то передача исключения (см. раздел 5.6) типа `regex_error` произойдет только *во время выполнения*. Подобно всем стандартным типам исключений, у исключения `regex_error` есть функция `what()`, описывающая произошедшую ошибку (см. раздел 5.6.2). У исключения `regex_error` есть также функция-член `code()`, возвращающая числовой код (зависящий от реализации), соответствующий типу произошедшей ошибки. Стандартные сообщения об ошибках, которые могут быть переданы библиотекой RE, приведены в табл. 17.7.

Таблица 17.7. Причины ошибок в регулярном выражении

Определены в типах <code>regex</code> и <code>regex_constants</code> : : <code>syntax_option_type</code>	
<code>error_collate</code>	Недопустимый запрос объединения элементов
<code>error_ctype</code>	Недопустимый класс символов
<code>error_escape</code>	Недопустимый управляемый или замыкающий символ
<code>error_backref</code>	Недопустимая обратная ссылка
<code>error_brack</code>	Несоответствие квадратных скобок ([или])
<code>error_paren</code>	Несоответствие круглых скобок ((или))
<code>error_brace</code>	Несоответствие фигурных скобок {{ или }})
<code>error_badbrace</code>	Недопустимый диапазон в фигурных скобках ({ })
<code>error_range</code>	Недопустимый диапазон символов (например, [z-a])

error_space	Недостаточно памяти для выполнения этого регулярного выражения
error_badrepeat	Повторяющийся символ (*?, + или {) не предваряется допустимым регулярным выражением
error_complexity	Затребованное соответствие слишком сложно
error_stack	Недостаточно памяти для вычисления соответствия

Например, в схеме вполне можно пропустить по неосторожности скобку:

```
try {
    // ошибка: пропущена закрывающая скобка после
alnum; конструктор
    // передаст исключение
    regex      r( "[[:alnum:]]+\\". (cpp|cxx|cc)$",
regex::icase);
} catch (regex_error e)
{
    cout << e.what() << "\ncode: " << e.code() <<
endl; }
```

При запуске на системе авторов эта программа выводит следующее:

```
regex_error(error_brack):
The expression contained mismatched [ and ].
code: 4
```

Компилятор определяет функцию-член `code()` для возвращения позиции ошибок, перечисленных в табл. 17.7, счет которых, как обычно, начинается с нуля.

Совет. Избегайте создания ненужных регулярных выражений

Как уже упоминалось, представляющая регулярное выражение "программа" компилируется во время выполнения, а не во время компиляции. Компиляция регулярного выражения может быть на удивление медленной операцией, особенно если используется расширенная грамматика регулярного выражения или выражение слишком сложно. В результате создание объекта класса `regex` и присвоение нового регулярного выражения уже существующему объекту класса `regex` может занять много времени. Для минимизации этих дополнительных затрат не создавайте больше объектов класса `regex`, чем необходимо. В частности, если регулярное выражение используются в цикле, его следует создать вне цикла, избежав перекомпиляции при

каждой итерации.

Классы регулярного выражения и тип исходной последовательности

Поиск возможен в любой из исходных последовательностей нескольких типов. Входные данные могут быть обычными символами типа `char` или `wchar_t`, и эти символы могут храниться в библиотечной строке или в массиве символов (или в версии для `wchar_t`, или `wstring`). Библиотека RE определяет отдельные типы, соответствующие этим разным типам исходных последовательностей.

Предположим, например, что класс `regex` содержит регулярное выражение типа `char`. Для типа `wchar_t` библиотека определяет также класс `wregex`, поддерживающий все операции класса `regex`. Единственное различие в том, что инициализаторы класса `wregex` должны использовать тип `wchar_t` вместо типа `char`.

Типы соответствий и итераторов (они рассматриваться в следующих разделах) более специфичны. Они отличаются не только типом символов, но и тем, является ли последовательность библиотечным типом или массивом: класс `smatch` представляет исходные последовательности типа `string`; класс `cmatch` — символьные массивы; `wsmatch` — строки Unicode (`wstring`); `wcmatch` — массивы символов `wchar_t`.

Таблица 17.8. Библиотечные классы регулярных выражений

Тип исходной последовательности	Используемый класс регулярного выражения
<code>string</code>	<code>regex, smatch, ssub_match И sregex_iterator</code>
<code>const char*</code>	<code>regex, cmatch, csub_match И cregex_iterator</code>
<code>wstring</code>	<code>wregex, wsmatch, wssub_match И wsregex_iterator</code>
<code>const wchar_t*</code>	<code>wregex, wcmatch, wcssub_match И wcregex_iterator</code>

Важный момент: используемый тип библиотеки RE должен соответствовать типу исходной последовательности. Соответствие классов видам исходных последовательностей приведено в табл. 17.8. Например:

```
regex r( "[[:alnum:]]+\.\.(cpp|cxx|cc)$",  
regex::icase);  
smatch results; // будет соответствовать  
последовательности типа
```

```
// string, но не char*
if (regex_search("myfile.cc", results, r)) // ошибка: ввод char*
    cout << results.str() << endl;
```

Компилятор C++ отклонит этот код, поскольку тип аргумента и тип исходной последовательности не совпадают. Если необходимо искать в символьном массиве, то следует использовать объект класса `cmatch`:

```
cmatch results; // будет соответствовать
последовательности символьного
// массива
if (regex_search("myfile.cc", results, r))
    cout << results.str() << endl; // вывод текущего
соответствия
```

Обычно программы используют исходные последовательности типа `string` и соответствующие ему версии компонентов библиотеки RE.

Упражнения раздела 17.3.1

Упражнение 17.14. Напишите несколько регулярных выражений, предназначенных для создания различных ошибок. Запустите программу и посмотрите, какие сообщения выводит ваш компилятор для каждой ошибки.

Упражнение 17.15. Напишите программу, используя схему поиска слов, нарушающих правило "*i* перед *e*, кроме как после *c*". Организуйте приглашение для ввода пользователем слова и вывод результата его проверки. Проверьте свою программу на примере слов, которые нарушают и не нарушают это правило.

Упражнение 17.16. Что будет при инициализации объекта класса `regex` в предыдущей программе значением "[^c] ei"? Проверьте свою программу, используя эту схему, и убедитесь в правильности своих ожиданий.

17.3.2. Типы итераторов классов соответствия и `regex`

Программа проверки правила "*i* перед *e*, кроме как после *c*" из раздела 17.3.1 выводила только первое соответствие в исходной последовательности. Используя итератор `sregex_iterator`, можно получить все соответствия. Итераторы класса `regex` являются адаптерами итератора (см. раздел 9.6), привязанные к исходной последовательности и объекту класса `regex`. Как было описано в табл. 17.8, для каждого типа исходной последовательности используется специфический тип итератора. Операции с итераторами описаны в табл. 17.9.

Когда итератор `sregex_iterator` связывается со строкой и объектом класса `regex`, итератор автоматически позиционируется на первое соответствие в заданной строке. Таким образом, конструктор `sregex_iterator()` вызывает функцию `regex_search()` для данной строки и объекта класса `regex`. При обращении к значению итератора возвращается объект класса `smatch`, соответствующий результатам самого последнего поиска. При приращении итератора для поиска следующего соответствия в исходной строке вызывается функция `regex_search()`.

Таблица 17.9. Операции с итератором `sregex_iterator`

Эти операции применимы также к итераторам <code>cregex_iterator</code> , <code>wsregex_iterator</code> и <code>wcregex_iterator</code>	
<code>sregex_iterator</code>	<code>it</code> — это итератор <code>sregex_iterator</code> , перебирающий строку, обозначенную итераторами <code>b</code> и <code>e</code> . Вызов <code>regex_search(b, e, r)</code> устанавливает итератор <code>it</code> на первое соответствие во вводе
<code>sregex_iterator end;</code>	Итератор <code>sregex_iterator</code> , указывающий на позицию после конца
<code>*it it-></code>	Возвращает ссылку на объект класса <code>smatch</code> или указатель на объект класса <code>smatch</code> от самого последнего вызова функции <code>regex_search()</code>
<code>++it it++</code>	Вызывает функцию <code>regex_search()</code> для исходной последовательности, начиная сразу после текущего соответствия. Префиксная версия возвращает ссылку на приращенный итератор, а постфиксная возвращает прежнее значение
<code>it1 == it2</code> <code>it1 != it2</code>	Два итератора <code>sregex_iterator</code> равны, если оба они итераторы после конца. Два не конечных итератора равны, если они созданы из той же исходной последовательности и объекта класса <code>regex</code>

Использование итератора sregex_iterator

В качестве примера дополним программу поиска нарушения правила "*i* перед *e*, кроме как после *c*" в текстовом файле. Подразумевается, что *file* класса *string* содержит все содержимое исходного файла, на котором осуществляется поиск. Новая версия программы будет использовать ту же схему, что и ранее, но для поиска применим итератор *sregex_iterator*:

```
// найти символы ei, следующие за любым символом,
// кроме с
string pattern("[ ^c] ei");
// искомая схема должна присутствовать в целом
// слове
pattern = "[[:alpha:]]*" + pattern + "[[:alpha:]]*";
regex r(pattern, regex::icase); // игнорируем
// случай выполнения
// соответствия
// будет последовательно вызывать regex_search()
// для поиска всех
// соответствий в файле
for (sregex_iterator it(file.begin(), file.end(), r), end_it;
     it != end_it; ++it)
    cout << it->str() << endl; // соответствующее
// слово
```

Цикл *for* перебирает все соответствия *r* в строке *file*. Инициализатор в цикле *for* определяет итераторы *it* и *end_it*. При определении итератора *it* конструктор *sregex_iterator()* вызывает функцию *regex_search()* для позиционирования итератора *it* на первое соответствие в строке *file*.

Пустой итератор *sregex_iterator*, *end_it* действует как итератор после конца. Приращение в цикле *for* "перемещает" итератор, вызвав функцию *regex_search()*. При обращении к значению итератора возвращается объект класса *smatch*, представляющий текущее соответствие. Для вывода соответствующего слова вызывается функция-член *str()*.

Данный цикл *for* как бы перепрыгивает с одного соответствия на другое, как показано на рис. 17.1.



Рис. 17.1. Использование итератора `sregex_iterator`

Использование данных соответствия

Если запустить этот цикл для строки `test_str` из первоначальной программы, вывод был бы таким:

```
freind
theif
```

Однако вывод только самого слова, соответствующего заданному выражению, не очень полезен. При запуске программы для большой исходной последовательности, например для текста этой главы, имело бы смысл увидеть контекст, в котором встретилось слово. Например:

```
hey read or write according to the type
      >>> being <<<
handled. The input operators ignore whi
```

Кроме возможности вывода части исходной строки, в которой встретилось соответствие, классы соответствия предоставляют более подробную информацию о соответствии. Возможные операции с этими типами перечислены в табл. 17.10 и 17.11.

Более подробная информация о `smatch` и `ssub_match` приведена в следующем разделе, а пока достаточно знать, что они предоставляют доступ к контексту соответствия. У типов соответствия есть функции-члены `prefix()` и `suffix()`, возвращающие объект класса `ssub_match`, представляющий часть исходной последовательности перед и после текущего соответствия соответственно. У класса `ssub_match` есть функции-члены `str()` и `length()`, возвращающие соответствующую строку и ее размер соответственно. Используя эти функции, можно переписать цикл программы проверки правописания:

```
// тот же заголовок цикла for, что и прежде
for (sregex_iterator it(file.begin(), file.end(), r), end_it;
     it != end_it; ++it) {
```

```

        auto pos = it->prefix().length(); // размер
предикса
        pos = pos > 40 ? pos - 40 : 0; // необходимо до
40 символов
        cout << it->prefix().str().substr(pos) // последняя часть предикса
        << "\n\t\t>>> " << it->str() << " <<<\n" // соответствующее
        //
СЛОВО
        << it->suffix().str().substr(0, 40) // первая
часть суффикса
        << endl;
}

```

Таблица 17.10. Операции с типом smatch

Эти операции применимы также к типам cmatch, wsmatch, wcmatch и соответствующим типам csub_match, wssub_match и wcsub_match.	
m. ready()	Возвращает значение <code>true</code> , если <code>m</code> был установлен вызовом функции <code>regex_search()</code> или <code>regex_match()</code> , в противном случае — значение <code>false</code> (в этом случае результат операции с <code>m</code> непредсказуем)
m. size()	Возвращает значение 0, если соответствия не найдено, в противном случае — на единицу больше, чем количество подвыражений в последнем соответствующем регулярном выражении
m. empty()	Возвращает значение <code>true</code> , если размер нулевой
m. prefix()	Возвращает объект класса <code>ssub_match</code> , представляющий последовательность перед соответствием
m. suffix()	Возвращает объект класса <code>ssub_match</code> , представляющий часть после конца соответствия
m. format(...)	См. табл. 17.12
В функциях, получающих индекс, <code>n</code> по умолчанию имеет значение нуль и должно быть меньше <code>m.size()</code>. Первое соответствие (с индексом 0) представляет общее соответствие.	
m. length(n)	Возвращает размер соответствующего подвыражения номер <code>n</code>
m. position(n)	Дистанция подвыражения номер <code>n</code> от начала последовательности
m. str(n)	Соответствующая строка для подвыражения номер <code>n</code>

m[n]	Объект <code>ssub_match</code> , соответствующий подвыражению номер n
<code>m.begin() , m.end()</code> <code>m.cbegin() , m.cend()</code>	Итераторы элементов <code>sub_match</code> в <code>m</code> . Как обычно, функции <code>cbegin()</code> и <code>cend()</code> возвращают итераторы <code>const_iterator</code>

Более подробная информация о `smatch` и `ssub_match` приведена в следующем разделе, а пока достаточно знать, что они предоставляют доступ к контексту соответствия. У типов соответствия есть функции-члены `prefix()` и `suffix()`, возвращающие объект класса `ssub_match`, представляющий часть исходной последовательности перед и после текущего соответствия соответственно. У класса `ssub_match` есть функции-члены `str()` и `length()`, возвращающие соответствующую строку и ее размер соответственно. Используя эти функции, можно переписать цикл программы проверки правописания:

```
// тот же заголовок цикла for, что и прежде
for (sregex_iterator it(file.begin(), file.end(), r), end_it;
     it != end_it; ++it) {
    auto pos = it->prefix().length(); // размер
    // префикса
    pos = pos > 40 ? pos - 40 : 0; // необходимо до
    // 40 символов
    cout << it->prefix().str().substr(pos) // последняя часть префикса
        << "\n\t\t>>> " << it->str() << " <<<\n" // соответствующее
    // СЛОВО
    << it->suffix().str().substr(0, 40) // первая
    // часть суффикса
    << endl;
}
```

Сам цикл работает, как и прежде. Изменился процесс в цикле `for`, представленный на рис. 17.2.

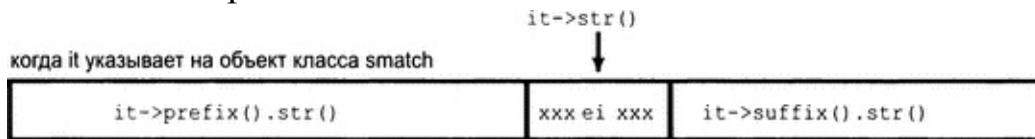


Рис. 17.2. Объект класса `smatch`, представляющий некое соответствие

Здесь происходит вызов функции `prefix()`, возвращающей объект класса `ssub_match`, представляющий часть строки `file` перед текущим соответствием. Чтобы выяснить, сколько символов находится в части строки `file` перед соответствием, вызовем функцию `length()` для этого объекта класса `ssub_match`. Затем скорректируем значение `pos` так, чтобы оно было индексом 40-го символа от конца префикса. Если у префикса меньше 40 символов, устанавливаем `pos` в 0, означая, что выведен весь префикс. Функция `substr()` (см. раздел 9.5.1) используется для вывода от данной позиции до конца префикса.

После вывода символов, предшествующих соответствуию, выводится само соответствие с некоторым дополнительным оформлением, чтобы соответствующее слово выделилось в выводе. После вывода соответствующей части выводится до 40 следующих после соответствия символов строки `file`.

Упражнения раздела 17.3.2

Упражнение 17.17. Измените свою программу так, чтобы она находила все слова в исходной последовательности, нарушающие правило "i перед e, кроме как после c".

Упражнение 17.18. Пересмотрите свою программу так, чтобы игнорировать слова, содержащие сочетание "ei", но не являющиеся ошибочными, такие как "albeit" и "neighbor".

17.3.3. Использование подвыражений

Схема в регулярном выражении зачастую содержит одно или несколько *подвыражений* (subexpression). Подвыражение — это часть схемы, которая сама имеет значение. Для обозначения подвыражения в регулярном выражении, как правило, используют круглые скобки.

Например, в схеме для поиска соответствий расширений файлов языка C++ (см. раздел 16.1.2) круглые скобки используются для группировки возможных расширений. Каждый раз, когда альтернативы группируются с использованием круглых скобок, одновременно объявляется, что эти альтернативы формируют подвыражение. Это выражение можно переписать так, чтобы оно предоставило доступ к имени файла, являющемуся той частью схемы, которая предшествует точке:

```
// r содержит два подвыражения:  
// первое - часть имени файла перед точкой,  
// второе - расширение файла  
regex r("( [[:alnum:] ]+) \\. (cpp|cxx|cc)$",  
regex::icase);
```

Теперь в схеме два заключенных в скобки подвыражения:

- (`[[:alnum:]]+`) — представляет последовательность из одного или нескольких символов;
- (`cpp|cxx|cc`) — представляет расширения файлов.

Теперь программу из раздела 16.1.2 можно переписать так (изменив оператора вывода), чтобы выводить только имя файла:

```
if (regex_search(filename, results, r))  
    cout << results.str(1) << endl; // вывести первое  
подвыражение
```

В первоначальной программе для поиска схемы `r` в строке `filename` использовался вызов функции `regex_search()`, а также объект `results` класса `smatch` для содержания результата поиска соответствия. Если вызов успешен, выводится результат. Но в этой программе выводится `str(1)`, т.е. соответствие для первого подвыражения.

Кроме информации об общем соответствии, объекты соответствия предоставляют доступ к каждому соответствию подвыражению в схеме. К соответствиям подвыражению обращаются по позиции. Первое соответствие подвыражению, расположенное в позиции 0, представляет соответствие для всей схемы. После него располагается каждое подвыражение. Следовательно, имя файла, являющееся первым

подвыражением в схеме, находится в позиции 1, а расширение файла — в позиции 2.

Например, если именем файла будет `foo.cpp`, то `results.str(0)` содержит строку `"foo.cpp"`; `results.str(1) — "foo"`, а `results.str(2) — "cpp"`.

В этой программе требуется часть имени перед точкой, что является первым подвыражением, поэтому следует вывести `results.str(1)`.

Подвыражения для проверки правильности данных

Подвыражения обычно используются для проверки данных, которые должны соответствовать некоему определенному формату. Например, в Америке номера телефонов имеют десять цифр, включая код города и местный номер из семи цифр. Код города зачастую, но не всегда, заключен в круглые скобки. Остальные семь цифр могут быть отделены тире, точкой или пробелом либо не отделяться вообще. Данные в некоторых из этих форматов могли бы быть приемлемы, а в других — нет. Процесс будет состоять из двух этапов: сначала используем регулярное выражение для поиска последовательностей, которые могли бы быть номерами телефонов, а затем вызовем функцию для окончательной проверки правильности данных.

Прежде чем написать схему номеров телефона, необходимо рассмотреть еще несколько аспектов языка регулярных выражений на языке ECMAScript.

- `\{ d\}` представляет одиночную цифру, а `\{ d\} { n}` — последовательность из `n` цифр. (Например, `\{ d\} { 3}` соответствует последовательности из трех цифр.)

- Набор символов в квадратных скобках позволяет задать соответствие любому из трех символов. (Например, `[-.]` соответствует тире, точке или пробелу. Обратите внимание: у точки в квадратных скобках нет никакого специального смысла.)

- Компонент, следующий за символом `' ?'`, не обязательный. (Например, `\{ d\} { 3} [-.] ? \{ d\} { 4}` соответствует трем цифрам, сопровождаемым опциональными тире, точкой или пробелом и еще четырьмя цифрами. Этой схеме соответствовало бы `555-0132`, или `555.0132`, или `555 0132`, или `5550132`).

- Как и в языке C++, в ECMAScript символ за наклонной чертой означает, что он представляет себя, а не специальное значение. Поскольку данная схема включает круглые скобки, являющиеся специальными

символами в языке ECMAScript, круглые скобки, являющиеся частью схемы, следует представить как \(или \) .

Поскольку наклонная черта влево является специальным символом в языке C++, когда он встречается в схеме, следует добавить вторую наклонную черту, чтобы указать языку C++, что имеется в виду символ \ . Следовательно, чтобы представить регулярное выражение \{ d\} { 3} , нужно написать \\{ d\} { 3} .

Для проверки номеров телефонов следует обратиться к компонентам схемы. Например, необходимо проверить, что если номер использует открывающую круглую скобку для кода города, то он использует также закрывающую скобку после него. В результате такой номер, как (908. 555. 1800 , следует отклонить.

Для определения такого соответствия необходимо регулярное выражение, использующее подвыражения. Каждое подвыражение заключается в пару круглых скобок:

```
// все выражение состоит из семи подвыражений:  
(ddd) разделитель ddd  
// разделитель dddd  
// подвыражения 1, 3, 4 и 6 optionalны; а 2, 5 и 7  
содержат цифры  
"(\\()?(\\d{3})(\\())?([-.])?(\\d{3})([.-])?  
(\\d{4})";
```

Поскольку схема использует круглые скобки, а также из-за использования наклонных черт, эту схему трудно прочитать (и написать!). Проще всего прочитать ее по каждомуциальному (заключенному в скобки) подвыражению.

1. (\\() ? необязательная открывающая скобка для кода города.
2. (\\d{3}) код города.
3. (\\()) ? необязательная закрывающая скобка для кода города.
4. ([-.]) ? необязательный разделитель после кода города.
5. (\\d{3}) следующие три цифры номера.
6. ([-.]) ? другой необязательный разделитель.
7. (\\d{4}) последние четыре цифры номера.

Следующий код использует эту схему для чтения файла и находит данные, соответствующие общей схеме телефонных номеров. Для проверки допустимости формата номеров используется функция valid() :

```
string phone =  
"(\\()?(\\d{3})(\\())?([.-])?(\\d{3})([.-])?)"
```

```

( \\d{ 4 } ) ";
    regex r( phone ); // объект regex для поиска схемы
    smatch m;
    string s;
    // прочитать все записи из входного файла
    while ( getline( cin, s ) ) {
        // для каждого подходящего номера телефона
        for ( sregex_iterator it( s.begin( ), s.end( ), r ),
end_it;
                it != end_it; ++it )
            // проверить допустимость формата номера
            if ( valid( *it ) )
                cout << "valid: " << it->str() << endl;
            else
                cout << "not valid: " << it->str() << endl;
    }
}

```

Операции с типом соотвествия

Напишем функцию `valid()`, используя операции типа соотвествия, приведенные в табл. 17.11. Не следует забывать, что схема `pattern` состоит из семи подвыражений. В результате каждый объект класса `smatch` будет содержать восемь элементов `ssub_match`. Элемент `[0]` представляет общее соотвествие, а элементы `[1] - [7]` представляют каждое из соотвествующих подвыражений.

Таблица 17.11. Операции с типом соотвествия

Эти операции применимы к типам <code>ssub_match</code>, <code>csub_match</code>, <code>wssub_match</code> и <code>wcsub_match</code>	
<code>matched</code>	Открытая логическая переменная-член, означающая соотвествие объекта класса <code>ssub_match</code>
<code>first</code> <code>second</code>	Открытые переменные-члены, являющиеся итераторами на начало последовательности соотвествия и ее следующий элемент после последнего. Если соотвествия нет, то <code>first</code> и <code>second</code> равны
<code>length()</code>	Размер текущего объекта соотвествия. Возвращает 0, если переменная-член <code>matched</code> содержит значение <code>false</code>
<code>str()</code>	Возвращает строку, содержащую соотвествующую часть ввода. Возвращает пустую строку, если переменная-член <code>matched</code> содержит значение <code>false</code>
	Преобразует объект <code>ssub</code> класса <code>ssub_match</code> в строку <code>s</code> . Эквивалент

`s = ssub` | вызова `s = ssub.str()`. Оператор преобразования не является явным
(см. раздел 14.9.1)

Когда происходит вызов функции `valid()`, известно, что общее соответствие имеется, но неизвестно, какие из необязательных подвыражений являются частью этого соответствия. Переменная-член `matched` класса `ssub_match`, соответствующая определенному подвыражению, содержит значение `true`, если это подвыражение является частью общего соответствия.

В правильном номере телефона код города либо полностью заключается в скобки, либо не заключается в них вообще. Поэтому действие функции `valid()` зависит от того, начинается ли номер с круглой скобки или нет:

```
bool valid(const smatch& m) {
    // если перед кодом города есть открывающая скобка
    if (m[1].matched)
        // за кодом города должна быть закрывающая скобка
        // и остальная часть номера непосредственно или
        // через пробел
        return m[3].matched
            && (m[4].matched == 0 || m[4].str() == "
");
    else
        // здесь после кода города не может быть
        // закрывающей скобки
        // но разделители между другими двумя
        // компонентами должны быть
        // корректны
        return !m[3].matched
            && m[4].str() == m[6].str();
}
```

Начнем с проверки соответствия первому подвыражению (т.е. открывающей скобки). Это подвыражение находится в элементе `m[1]`. Если это соответствие есть, то номер начинается с открывающей скобки. В таком случае номер будет допустимым, только если подвыражение после кода города также будет соответствующим (т.е. будет закрывающая скобка после кода города). Кроме того, если скобки в начале номера корректны, то следующим символом должен быть пробел или первая цифра следующей части номера.

Если элемент `m[1]` не соответствует (т.е. открывающей скобки нет), то подвыражение после кода города также должно быть пустым. Если это так и если остальные разделители совпадают, то номер допустим, но не в противном случае.

Упражнения раздела 17.3.3

Упражнение 17.19. Почему можно вызывать функцию `m[4].str()` без предварительной проверки соответствия элемента `m[4]`?

Упражнение 17.20. Напишите собственную версию программы для проверки номеров телефонов.

Упражнение 17.21. Перепишите программу номеров телефонов из раздела 8.3.2 так, чтобы использовать функцию `valid()`, определенную в этом разделе.

Упражнение 17.22. Перепишите программу номеров телефонов так, чтобы она позволила разделять три части номера телефона любыми символами.

Упражнение 17.23. Напишите регулярное выражение для поиска почтовых индексов. У них может быть пять или девять цифр. Первые пять цифр могут быть отделены от остальных четырех тире.

17.3.4. Использование функции `regex_replace()`

Регулярные выражения зачастую используются не только для поиска, но и для замены одной последовательности другой. Например, может потребоваться преобразовать американские номера телефонов в формат "ddd.ddd.dddd", где код города и три последующие цифры разделены точками.

Когда необходимо найти и заменить регулярное выражение в исходной последовательности, используется функция `regex_replace()`. Подобно функции поиска, функция `regex_replace()`, описанная в табл. 17.12, получает входную символьную последовательность и объект класса `regex`. Следует также передать строку, которая описывает необходимый вывод.

Таблица 17.12. Функции замены регулярного выражения

	Создает форматированный вывод, используя формат строки <code>fmt</code> , соответствие в <code>m</code> и необязательные флаги <code>match_flag_type</code> в <code>mft</code> .
--	--

```
m.format(dest, fmt, mft)  
m.format(fmt, mft)
```

Первая версия пишет в итератор вывода `dest` (см. раздел 10.5.1) и получает формат `fmt`, который может быть строкой или парой указателей, обозначающих диапазон в символьном массиве. Вторая версия возвращает строку, которая содержит вывод и получает формат `fmt`, являющийся строкой или указателем на символьный массив с нулевым символом в конце. По умолчанию `mft` имеет значение `format_default`

```
regex_replace(dest, seq, r, fmt, mft)  
regex_replace(seq, r, fmt, mft)
```

Перебирает последовательность `seq`, используя функцию `regex_search()` для поиска соответствий объекту `r` класса `regex`. Использует формат строки `fmt` и необязательные флаги `match_flag_type` в `mft` для формирования вывода. Первая версия пишет в итератор вывода `dest` и получает пару итераторов для обозначения последовательности `seq`. Вторая возвращает строку, содержащую вывод, а `seq` может быть строкой или указателем на символьный массив с нулевым символом в конце. Во всех случаях формат `fmt` может быть строкой или указателем на символьный массив с нулевым символом в конце. По умолчанию `mft` имеет значение `match_default`

Строку замены составляют подлежащие включению символы вместе с подвыражениями из соответствующей подстроки. В данном случае следует использовать второе, пятое и седьмое подвыражения из строки замены. Первое, третье, четвертое и шестое подвыражения игнорируются, поскольку они использовались в первоначальном форматировании номера, но не являются частью формата замены. Для ссылки на конкретное подвыражение используется символ `$`, сопровождаемый индексом подвыражения:

```
string fmt = "$2.$5.$7"; // переформатировать  
номера в ddd.ddd.ddd
```

Схему регулярного выражения и строку замены можно использовать следующим образом:

```
regex r(phone); // regex для поиска схемы
```

```
string number = "( 908 ) 555-1800";
cout << regex_replace( number, r, fmt) << endl;
Вывод этой программы будет таким:
908.555.1800
```

Замена только части исходной последовательности

Куда интересней использование обработки регулярных выражений для замены номеров телефонов в большом файле. Предположим, например, что имеется файл имен и номеров телефонов, содержащий такие данные:

```
morgan ( 201 ) 555-2368 862-555-0123/
drew ( 973 ) 555.0130
lee ( 609 ) 555-0132 2015550175 800.555-0000
```

Их следует преобразовать в такой формат:

```
morgan 201.555.2368 862.555.0123
drew 973.555.0130
lee 609.555.0132 201.555.0175 800.555.0000
```

Это преобразование можно осуществить следующим образом:

```
int main() {
    string phone =
        "( \\\()?( \\\d{ 3 })( \\))?( [ -. ] )?( \\\d{ 3 })( [ -. ] )?
        ( \\\d{ 4 }) ";
    regex r( phone ); // regex для поиска схемы
    smatch m;
    string s;
    string fmt = "$2.$5.$7"; // перепрограммировать
    // номера в ddd.ddd.ddd
    // прочитать каждую запись из входного файла
    while ( getline( cin, s ) )
        cout << regex_replace( s, r, fmt ) << endl;
    return 0;
}
```

Каждая запись читается в строку `s` и передается функции `regex_replace()`. Эта функция находит и преобразует все соответствия исходной последовательности.

Флаги, контролирующие соответствие и формат

Кроме флагов обработки регулярных выражений, библиотека определяет также флаги, позволяющие контролировать процесс поиска соответствия и форматирования при замене. Их значения приведены в

табл. 17.13. Эти флаги могут быть переданы функции `regex_search()`, или функции `regex_match()`, или функциям-членам формата класса `smatch`.

Таблица 17.13. Флаги соответствия

Определено в <code>regex_constants::match_flag_type</code>	
<code>match_default</code>	Эквивалент <code>format_default</code>
<code>match_not_bol</code>	Не рассматривать первый символ как начало строки
<code>match_not_eol</code>	Не рассматривать последний символ как конец строки
<code>match_not_bow</code>	Не рассматривать первый символ как начало слова
<code>match_not_eow</code>	Не рассматривать последний символ как конец слова
<code>match_any</code>	Если соответствий несколько, может быть возвращено любое из них
<code>match_not_null</code>	Не соответствует пустой последовательности
<code>match_continuous</code>	Соответствие должно начинаться с первого символа во вводе
<code>match_prev_avail</code>	У исходной последовательности есть символы перед первым
<code>format_default</code>	Строка замены использует правила ECMAScript
<code>format_sed</code>	Строка замены использует правила POSIX <code>sed</code>
<code>format_no_copy</code>	Не выводить несоответствующие части ввода
<code>format_first_only</code>	Заменить только первое вхождение

Флаги соответствия и формата имеют тип `match_flag_type`. Их значения определяются в пространстве имен `regex_constants`. Подобно пространству имен `placeholders`, используемому с функциями `bind()` (см. раздел 10.3.4), пространство имен `regex_constants` определено в пространстве имен `std`. Для использования имени из пространства `regex_constants` его следует квалифицировать именами обоих пространств имен:

```
using std::regex_constants::format_no_copy;
```

Это объявление указывает, что когда код использует флаг `format_no_copy`, необходим объект из пространства имен `std::regex_constants`. Вместо этого можно использовать и альтернативную форму `using`, рассматриваемую в разделе 18.2.2:

```
using namespace std::regex_constants;
```

Использование флагов формата

По умолчанию функция `regex_replace()` выводит всю исходную

последовательность. Части, которые не соответствуют регулярному выражению, выводятся без изменений, а соответствующие части оформляются, как указано строкой формата. Это стандартное поведение можно изменить, указав флаг `format_no_copy` в вызове функции `regex_replace()`:

```
// выдать только номера телефона: используется  
новая строка формата  
string fmt2 = "$2.$5.$7 "; // поместить пробел как  
разделитель после // последнего числа  
// указать regex_replace() копировать только  
заменяемый текст  
cout << regex_replace(s, r, fmt2, format_no_copy)  
<< endl;
```

С учетом того же ввода эта версия программы создает такой вывод:

```
201.555.2368 862.555.0123  
973.555.0130  
609.555.0132 201.555.0175 800.555.0000
```

Упражнения раздела 17.3.4

Упражнение 17.24. Напишите собственную версию программы для переформатирования номеров телефонов.

Упражнение 17.25. Перепишите свою программу телефонных номеров так, чтобы она выводила только первый номер для каждого человека.

Упражнение 17.26. Перепишите свою программу телефонных номеров так, чтобы она выводила только второй и последующие номера телефонов для людей с несколькими номерами телефонов.

Упражнение 17.27. Напишите программу, которая переформатировала бы почтовый индекс с девятью цифрами как dddd-dddd.

17.4. Случайные числа

C++
11

Программы нередко нуждаются в источнике случайных чисел. До нового стандарта языки C и C++ полагались на простую библиотечную функцию языка C по имени `rand()`. Эта функция создает псевдослучайные целые числа, равномерно распределенные в диапазоне от нуля до зависимого от системы максимального значения, которое по крайней мере не меньше 32767.

У функции `rand()` несколько проблем: многим, если не всем, программам нужны случайные числа в совершенно другом диапазоне, отличном от используемого функцией `rand()`. Некоторые приложения требуют случайных чисел с плавающей запятой, другим нужны числа с неоднородным распределением. Когда разработчики пытаются преобразовывать диапазон, тип или распределение чисел, созданных функцией `rand()`, их случайность зачастую теряется.

Библиотека случайных чисел, определенная в заголовке `random`, решает эти проблемы за счет набора взаимодействующих классов: классов *процессора случайных чисел* (*random-number engine*) и классов *распределения случайного числа* (*random-number distribution*). Эти классы описаны в табл. 17.14. Процессор создает последовательность беззнаковых случайных чисел, а распределение использует процессор для создания случайных чисел определенного типа в заданном диапазоне, распределенном согласно указанному вероятностному распределению.

Таблица 17.14. Компоненты библиотеки случайных чисел

Процессор	Типы, создающие последовательность случайных беззнаковых целых чисел
Распределение	Типы, использующие процессор для возвращения чисел согласно заданному распределению вероятности

Рекомендую

Программы C++ больше не должны использовать библиотечную функцию `rand()`. Для этого следует использовать класс `default_random_engine` наряду с соответствующим объектом

распределения.

17.4.1. Процессоры случайных чисел и распределения

Процессоры случайных чисел — это классы объектов функции (см. раздел 14.8), определяющие оператор вызова, не получающий никаких аргументов и возвращающий случайное беззнаковое число. Вызвав объект типа процессора случайных чисел, можно получить простые случайные числа:

```
default_random_engine e; // создает случайное  
беззнаковое число  
for (size_t i = 0; i < 10; ++i)  
    // e() "вызывает" объект для создания следующего  
    случайного числа  
    cout << e() << " ";
```

На системе авторов эта программа выводит:

```
16807 282475249 1622650073 984943658 1144108930  
470211272 ...
```

Здесь был определен объект `e` типа `default_random_engine`. В цикле `for` происходит вызов объекта `e`, возвращающий следующее случайное число.

Библиотека определяет несколько процессоров случайных чисел, отличающихся производительностью и качеством случайности. Каждый компилятор определяет один из этих процессоров как *стандартный процессор случайных чисел* (`default_random_engine`) (тип `default_random_engine`). Этот тип предназначен для процессоров с наиболее общеприменимыми свойствами (табл. 17.15). Список типов и функций процессоров, определенных стандартом, приведен в разделе А.3.2.

В большинстве случаев вывод процессора сам по себе непригоден для использования, поскольку, как уже упоминалось, это простые случайные числа. Проблема в том, что эти числа обычно охватывают диапазон, отличный от необходимого. *Правильное* преобразование диапазона случайного числа на удивление трудно.

Типы распределения и процессоры

Чтобы получить число в определенном диапазоне, используется объект типа распределения:

```
// однородное распределение от 0 до 9 включительно  
uniform_int_distribution<unsigned> u(0, 9);
```

```
default_random_engine e; // создает случайные  
беззнаковые целые числа  
for (size_t i = 0; i < 10; ++i)  
    // и использует e как источник чисел  
    // каждый вызов возвращает однородно  
распределенное значение  
    // в заданном диапазоне  
    cout << u(e) << " ";
```

Вывод таков:

```
0 1 7 4 5 2 0 6 6 9
```

Здесь `u` определяется как объект типа `uniform_int_distribution<unsigned>`. Этот тип создает однородно распределенные беззнаковые значения. При определении объекта этого типа можно задать minimum и maximum необходимых значений. Определение `u(0, 9)` указывает, что необходимы числа в диапазоне от 0 до 9 *включительно*. Распределение случайного числа использует включающие диапазоны, позволяющие получить любое возможное целочисленное значение в нем.

Подобно типам процессоров, типы распределения также являются классами объектов функции. Типы распределения определяют оператор вызова, получающий процессор случайных чисел как аргумент. Объект распределения использует свой аргумент процессора для создания случайного числа, которое объект распределения сопоставит с определенным распределением.

Обратите внимание на то, что объект процессора передается непосредственно, `u(e)`. Если бы вызов был написан как `u(e())`, то произошла бы попытка передать следующее созданное `e` значение в `u`, что привело бы к ошибке при компиляции. Поскольку некоторые распределения вызывают процессор несколько раз, передается сам процессор, а не очередной результат его вызова.



Когда упоминается *генератор случайных чисел* (random-number generator), имеется в виду комбинация объекта распределения с объектом процессора.

Сравнение процессора случайных чисел и функции rand()

Читатели, знакомые с библиотечной функцией `rand()` языка С, вероятно заметили, что вывод вызова объекта `default_random_engine` подобен выводу функции `rand()`. Процессоры предоставляют целые беззнаковые числа в определенном системой диапазоне. Функция `rand()` имеет диапазон от 0 до `RAND_MAX`. Диапазон процессора возвращается при вызове функций-членов `min()` и `max()` объекта его типа:

```
cout << "min: " << e.min() << " max: " << e.max()
<< endl;
```

На системе авторов эта программа выводит следующее

```
min: 1 max: 2147483646
```

Таблица 17.15. Операции с процессором случайного числа

<code>Engine e;</code>	Стандартный конструктор; использует заданное по умолчанию начальное число для типа процессора
<code>Engine e(s);</code>	Использует как начальное число целочисленное значение <code>s</code>
<code>e.seed(s)</code>	Переустанавливает состояние процессора, используя начальное число <code>s</code>
<code>e.min() e.max()</code>	Наименьшие и наибольшие числа, создаваемые данным генератором
<code>Engine::result_type</code>	Целочисленный беззнаковый тип, создаваемый данным процессором
<code>e.discard(u)</code>	Перемещает процессор на <code>u</code> шагов; <code>u</code> имеет тип <code>unsigned long long</code>

Процессоры создают последовательности чисел

У генераторов случайных чисел есть одно свойство, которое зачастую вызывает сомнения у новичков: даже при том, что создаваемые числа случайны, при каждом запуске данный генератор возвращает ту же последовательность чисел. Факт неизменности последовательности очень полезен во время проверки. С другой стороны, разработчики, использующие генераторы случайных чисел, должны учитывать этот факт.

Предположим, например, что необходима функция, создающая вектор из 100 случайных целых чисел, равномерно распределенных в диапазоне от 0 до 9. Могло бы показаться, что эту функцию следует написать следующим образом:

```
// безусловно неправильный способ создания
// вектора случайных целых чисел
// эта функция выводит те же 100 чисел при каждом
// вызове!
```

```
vector<unsigned> bad_randVec() {
    default_random_engine e;
    uniform_int_distribution<unsigned> u(0, 9);
    vector<unsigned> ret;
    for (size_t i = 0; i < 100; ++i)
        ret.push_back(u(e));
    return ret;
}
```

Однако при каждом вызове эта функция возвратит тот же вектор:

```
vector<unsigned> v1(bad_randVec());
vector<unsigned> v2(bad_randVec());
// выводит equal
cout << ((v1 == v2) ? "equal" : "not equal") <<
endl;
```

Этот код выводит "equal", поскольку векторы v1 и v2 имеют те же значения.

Для правильного написания этой функции объекты процессора и распределения следует сделать статическими (см. раздел 6.1.1):

```
// возвращает вектор из 100 равномерно
распределенных случайных чисел
vector<unsigned> good_randVec() {
    // поскольку процессоры и распределения хранят
    состояния, их следует
    // сделать статическими, чтобы при каждом вызове
    создавались новые
    // числа
    static default_random_engine e;
    static uniform_int_distribution<unsigned> u(0, 9);
    vector<unsigned> ret;
    for (size_t i = 0; i < 100; ++i)
        ret.push_back(u(e));
    return ret;
}
```

Поскольку объекты e и u являются статическими, они хранят свое состояние на протяжении вызовов функции. Первый вызов будет использовать первые 100 случайных чисел из последовательности, созданной вызовом u(e), а второй вызов создаст следующие 100 чисел и т.д.



Каждый генератор случайных чисел всегда создает ту же последовательность чисел. Функция с локальным генератором случайных чисел должна сделать объекты процессора и распределения статическими. В противном случае функция будет создавать ту же последовательность при каждом вызове.

Начальное число генератора

Тот факт, что генератор возвращает ту же последовательность чисел, полезен во время отладки. Но после проверки программы необходимо заставить ее создавать разные случайные результаты при каждом запуске. Для этого предоставляется *начальное число* (seed). Начальное число — это значение, которое процессор может использовать для начала создания чисел с нового пункта в последовательности.

Начальное число генератора можно задать одним из двух способов: предоставить его при создании объекта процессора либо вызвать функцию-член `seed()` класса процессора:

```
default_random_engine e1; // использует стандартное
начальное число
default_random_engine e2( 2147483646 ); // использует
зданное значение
// начального
числа
// e3 и e4 создадут ту же последовательность,
// поскольку они используют то же начальное число
default_random_engine e3; // использует стандартное
начальное число
e3.seed( 32767 ); // вызывает функцию seed() для
установки нового
// значения начального числа
default_random_engine e4( 32767 ); // устанавливает
начальное число 32767
for (size_t i = 0; i != 100; ++i) {
    if (e1() == e2())
        cout << "unseeded match at iteration: " << i <<
endl;
```

```
if ( e3() != e4() )
    cout << "seeded differs at iteration: " << i <<
endl;
```

Здесь определены четыре процессора. Первые два, `e1` и `e2`, имеют разные начальные числа и *должны* создавать разные последовательности. У двух вторых, `e3` и `e4`, то же значение начального числа. Эти два объекта создадут ту же последовательность.

Выбор подходящего начального числа, как и почти все при создании хороших наборов случайных чисел, на удивление сложен. Вероятно, наиболее распространен подход вызова системной функции `time()`. Эта функция, определенная в заголовке `ctime`, возвращает количество секунд, начиная с заданной эпохи. Функция `time()` получает один параметр, являющийся указателем на структуру для записи времени. Если этот указатель нулевой, функция только возвращает время:

```
default_random_engine e1( time( 0 ) ); // почти
случайное начальное число
```

Поскольку функция `time()` возвращает время как количество секунд, такое начальное число применимо только для приложений, создающих начальное число на уровне секунд или больших интервалов.



Функция `time()` обычно не используется как источник начального числа, если программа многократно запускается как часть автоматизированного процесса, поскольку она могла бы быть запущена с тем же начальным числом несколько раз.

Упражнения раздела 17.4.1

Упражнение 17.28. Напишите функцию, создающую и возвращающую равномерно распределенную последовательность случайных беззнаковых целых чисел при каждом вызове.

Упражнение 17.29. Позвольте пользователю предоставлять начальное число как необязательный аргумент функции, написанной в предыдущем упражнении.

Упражнение 17.30. Снова пересмотрите предыдущую функцию, позволив ей получать минимальное и максимальное значения для возвращаемых случайных чисел.

17.4.2. Другие виды распределений

Процессоры создают беззнаковые числа, и у каждого числа в диапазоне процессора есть та же вероятность быть созданным. Приложения зачастую нуждаются в числах других типов или распределений. Библиотека удовлетворяет обе эти потребности, определяя различные классы распределений, которые, будучи использованы с процессором, дают желаемый результат. Список операций, поддерживаемых типами распределения, приведен в табл. 17.16.

Таблица 17.16. Операции с распределениями

<i>Dist d;</i>	Стандартный конструктор; создает объект <i>d</i> готовым к использованию. Другие конструкторы зависят от типа <i>Dist</i> ; см. раздел А.3. Конструкторы распределений являются явными (см. раздел 7.5.4)
<i>d(e)</i>	Последовательные вызовы с тем же объектом <i>e</i> создадут последовательность случайных чисел согласно типу распределения <i>d</i> ; <i>e</i> — объект процессора случайных чисел
<i>d. min()</i> <i>d. max()</i>	Возвращает наименьшее и наибольшее числа, создаваемые <i>d(e)</i>
<i>d. reset()</i>	Восстанавливает состояние объекта <i>d</i> , чтобы последующее его использование не зависело от уже созданных значений

Создание случайных вещественных чисел

Программы нередко нуждаются в источнике случайных значений с плавающей точкой. В частности, в диапазоне от нуля до единицы.

Наиболее распространен *неправильный* способ получения случайного числа с плавающей точкой из функции *rand()* за счет деления результата ее выполнения на значение *RAND_MAX*, являющееся заданным системой верхним пределом случайного числа, возвращаемого функцией *rand()*. Этот подход *неправильный* потому, что у случайных целых чисел обычно меньшая точность, чем у чисел с плавающей запятой, поэтому некоторые значения с плавающей точкой никогда не будут получены.

Новые библиотечные средства позволяют легко получить случайное число с плавающей точкой. Достаточно определить объект типа *uniform_real_distribution* и позволить библиотеке соотнести случайные целые числа с числами с плавающей запятой. Подобно типу *uniform_int_distribution*, здесь также можно задать минимальные и максимальные значения при определении объекта:

```

default_random_engine e; // создает случайные
беззнаковые целые числа
// однородное распределение от 0 до 1 включительно
uniform_real_distribution<double> u(0,1);
for (size_t i = 0; i < 10; ++i)
    cout << u(e) << " ";

```

Этот код почти идентичен предыдущей программе, которая создавала беззнаковые значения. Но поскольку здесь использован другой тип распределения, данная версия дает другие результаты:

```

0.131538 0.45865 0.218959 0.678865 0.934693
0.519416 ...

```

Использование типа по умолчанию для результата распределения

За одним исключением, рассматриваемым в разделе 17.4.2, типы распределения являются шаблонами с одним параметром типа шаблона, представляющим тип создаваемых распределением чисел. Эти типы всегда создают либо тип с плавающей точкой, либо целочисленный тип.

У каждого шаблона распределения есть аргумент шаблона по умолчанию (см. раздел 16.1.3). Типы распределения, создающие значения с плавающей точкой, по умолчанию создают значения типа `double`. Распределения, создающие целочисленные результаты, используют по умолчанию тип `int`. Поскольку у типов распределения есть только один параметр шаблона, при необходимости использовать значение по умолчанию следует не забыть расположить за именем шаблона пустые угловые скобки, чтобы указать на применение типа по умолчанию (см. раздел 16.1.3):

```

// пустые <> указывают на использование
// для результата типа по умолчанию
uniform_real_distribution<> u(0,1); // по умолчанию
double

```

Создание чисел с неравномерным распределением

Кроме корректного создания случайных чисел в заданном диапазоне, новая библиотека позволяет также получить числа, распределенные неравномерно. Действительно, библиотека определяет 20 типов распределений! Эти типы перечисляются в разделе А.3.

Для примера создадим серию нормально распределенных значений и нарисуем полученное распределение. Поскольку тип

`normal_distribution` создает числа с плавающей запятой, данная программа будет использовать функцию `lround()` из заголовка `cmath` для округления каждого результата до ближайшего целого числа. Создадим 200 чисел с центром в значении 4 и среднеквадратичным отклонением 1,5. Поскольку используется нормальное распределение, можно ожидать любых чисел, но приблизительно 1% из них будет в диапазоне от 0 до 8 включительно. Программа подсчитает, сколько значений соответствует каждому целому числу в этом диапазоне:

```
default_random_engine e; // создает
случайные целые числа
normal_distribution<> n( 4, 1.5 ); // середина 4,
среднеквадратичное // отклонение 1.5
vector<unsigned> vals( 9 ); // девять элементов
со значением 0
for ( size_t i = 0; i != 200; ++i ) {
    unsigned v = lround( n( e ) ); // округление до
ближайшего целого
    if ( v < vals.size() ) // если результат в
диапазоне
        ++vals[ v ]; // подсчитать, как часто встречается
каждое число
}
for ( size_t j = 0; j != vals.size(); ++j )
    cout << j << ":" << string( vals[ j ], '*' ) << endl;
```

Начнем с определения объектов генератора случайных чисел и вектора `vals`. Вектор `vals` будет использован для расчета частоты создания каждого числа в диапазоне 0...9. В отличие от большинства других программ, использующих вектор, создадим его сразу с необходимым размером. Так, каждый его элемент инициализируется значением 0.

В цикле `for` происходит вызов функции `lround(n(e))` для округления возвращенного вызовом `n(e)` значения до ближайшего целого числа. Получив целое число, соответствующее случайному числу с плавающей точкой, используем его для индексирования вектора счетчиков. Поскольку вызов `n(e)` может создавать числа и вне диапазона от 0 до 9, проверим полученное число на принадлежность диапазону прежде, чем использовать его для индексирования вектора `vals`. Если число принадлежит диапазону, увеличиваем соответствующий счетчик.

Когда цикл заканчивается, вывод содержимого вектора `vals` выглядит следующим образом:

```
0: ***
1: *****
2: ***** ****
3: ***** **** ***** ****
4:
***** **** ***** **** ***** **** ***** **** ***** ****
5: ***** **** ***** **** ***** **** ***** **** ****
6: ***** **** **** **** ****
7: *****
8: *
```

Выведенные строки содержат столько звездочек, сколько раз встретилось соответствующее значение, созданное генератором случайных чисел. Обратите внимание: эта фигура не совершенно симметрична. Если бы она была симметрична, то возникли бы подозрения в качестве генератора случайных чисел.

Класс `bernoulli_distribution`

Как уже упоминалось, есть одно распределение, которое не получает параметр шаблона. Это распределение `bernoulli_distribution`, являющееся обычным классом, а не шаблоном. Это распределение всегда возвращает логическое значение `true` с заданной вероятностью. По умолчанию это вероятность `.5`.

В качестве примера распределения этого вида напишем программу, которая играет с пользователем. Игру начинает один из игроков (пользователь или программа). Чтобы выбрать первого игрока, можно использовать объект класса `uniform_int_distribution` с диапазоном от 0 до 1. В качестве альтернативы этот выбор можно сделать, используя распределение Бернулли. С учетом, что игру начинает функция `play()`, для взаимодействия с пользователем может быть использован следующий цикл:

```
string resp;
default_random_engine e; // e имеет состояние,
поэтому располагается
                        // вне цикла!
bernoulli_distribution b; // по умолчанию четность
50/50
do {
```

```

bool first = b( e ); // если true, программа ходит
первой
cout << (first ? "We go first"
           : "You get to go first") << endl;
// играть в игру, угадывая, кто ходит первым
cout << ((play(first)) ? "sorry, you lost"
           : "congrats, you won") << endl;
cout << "play again? Enter 'yes' or 'no'" << endl;
} while (cin >> resp && resp[ 0 ] == 'y' );

```

Для повторного запроса на продолжение игры используем цикл `do while` (см. раздел 5.4.4).



Поскольку процессоры возвращают ту же последовательность чисел (см. раздел 17.4.1), их объявляют за пределами циклов. В противном случае при каждой итерации создавался бы новый процессор, выдающий каждый раз те же значения. Распределения также могут хранить состояние и также должны определяться вне циклов.

Одна из причин использования в этой программе распределения `bernoulli_distribution` заключается в том, что это предоставит программе лучший шанс пойти первой:

```

bernoulli_distribution b(.55); // предоставить
программе небольшое
// преимущество

```

Такое определение `b` предоставит программе 55/45 шансов на первый ход.

Упражнения раздела 17.4.2

Упражнение 17.31. Что случилось бы в программе игры данного раздела, будь объекты `b` и `e` определены в цикле `do`?

Упражнение 17.32. Что случилось бы, будь строка `resp` определена в цикле?

Упражнение 17.33. Напишите версию программы преобразования слова из раздела 11.3.6, допускающую несколько преобразований для заданного слова и случайно выбирающую применяемое преобразование.

17.5. Еще о библиотеке ввода и вывода

Глава 8 познакомила вас с базовой архитектурой и наиболее часто используемой частью библиотеки ввода-вывода. В этом разделе рассматриваются три более специализированных средства, поддерживаемых библиотекой ввода-вывода: управление форматом, неформатированный ввод-вывод и произвольный доступ.

17.5.1. Форматированный ввод и вывод

Кроме флага состояния (см. раздел 8.1.2), каждый объект `iostream` имеет также флаг формата, контролирующий подробности формата ввода и вывода. Флаг формата контролирует такие аспекты, как формат записи целочисленных значений, точность значений с плавающей запятой, ширина выводимого элемента и т.д.

Библиотека определяет набор перечисленных в табл. 17.17 и 17.18 манипуляторов (manipulator) (см. раздел 1.2), изменяющих флаг формата потока. Манипулятор — это функция или объект, влияющие на состояние потока и применяемые как операнд оператора ввода или вывода. Как и операторы ввода и вывода, манипулятор возвращает потоковый объект, к которому он применяется; таким образом, можно объединить манипуляторы и данные в один оператор.

Таблица 17.17. Манипуляторы, определенные в объекте `iostream`

<code>boolalpha</code>	Отображать значения <code>true</code> и <code>false</code> как строки
<code>* noboolalpha</code>	Отображать значения <code>true</code> и <code>false</code> как 0 и 1
<code>showbase</code>	Создавать префикс, означающий базу целочисленных значений
<code>* noshowbase</code>	Не создавать префикс базы чисел
<code>showpoint</code>	Всегда отображать десятичную точку для значений с плавающей запятой
<code>* noshowpoint</code>	Отображать десятичную точку, только если у значения есть дробная часть
<code>showpos</code>	Отображать + для положительных чисел
<code>* noshowpos</code>	Не отображать + в неотрицательных числах
<code>uppercase</code>	Выводить 0x в шестнадцатеричной и E в экспоненциальной формах записи
<code>* nouppercase</code>	Выводить 0x в шестнадцатеричной и e в экспоненциальной формах записи
<code>* dec</code>	Отображать целочисленные значения с десятичной базой числа
<code>hex</code>	Отображать целочисленные значения с шестнадцатеричной базой числа
<code>oct</code>	Отображать целочисленные значения с восьмеричной базой числа
<code>left</code>	Добавлять дополняющие символы справа от значения
<code>right</code>	Добавлять дополняющие символы слева от значения

<code>internal</code>	Добавлять дополняющие символы между знаком и значением
<code>fixed</code>	Отображать значения с плавающей точкой в десятичном представлении
<code>scientific</code>	Отображать значения с плавающей точкой в экспоненциальном представлении
<code>hexfloat</code>	Отображать значения с плавающей точкой в шестнадцатеричном представлении (нововведение C++11)
<code>defaultfloat</code>	Вернуть формат числа с плавающей точкой в десятичный (нововведение C++11)
<code>unitbuf</code>	Сбрасывать буфер после каждой операции вывода
<code>*nounitbuf</code>	Восстановить обычный сброс буфера
<code>*skipws</code>	Пропускать отступы в операторах ввода
<code>noskipws</code>	Не пропускать отступы в операторах ввода
<code>flush</code>	Сбросить буфер объекта <code>ostream</code>
<code>ends</code>	Вставить нулевой символ, а затем сбросить буфер объекта <code>ostream</code>
<code>endl</code>	Вставить новую строку, а затем сбросить буфер объекта <code>ostream</code>

*Означает стандартное состояние потока

Таблица 17.18. Манипуляторы, определенные в объекте `iomanip`

<code>setfill(ch)</code>	Заполнить отступ символом <code>ch</code>
<code>setprecision(n)</code>	Установить точность <code>n</code> числа с плавающей точкой
<code>setw(w)</code>	Читать или писать значение в <code>w</code> символов
<code>setbase(b)</code>	Вывод целых чисел с базой <code>b</code>

Ранее в программах уже использовался манипулятор `endl`, который "записывался" в поток вывода как будто это значение. Но манипулятор `endl` — не обычное значение; он выполняет операцию: выводит символ новой строки и сбрасывает буфер.

Большинство манипуляторов изменяет флаг формата

Манипуляторы используются для двух общих категорий управления выводом: контроль представления числовых значений, а также контроль количества и расположения заполнителей. Большинство манипуляторов, изменяющих флаг формата, предоставлены парами для установки и сброса; один манипулятор устанавливает флаг формата в новое значение, а другой сбрасывает его, восстанавливая стандартное значение.



Манипуляторы, изменяющие флаг формата потока, обычно оставляют флаг формата измененным для всего последующего ввода-вывода.

Тот факт, что манипулятор вносит постоянное изменение во флаг формата, может оказаться полезным, когда имеется ряд операций ввода-вывода, использующих одинаковое форматирование. Действительно, некоторые программы используют эту особенность манипуляторов для изменения поведения одного или нескольких правил форматирования ввода или вывода. В таких случаях факт изменения потока является желательным.

Но большинство программ (и что еще важней, разработчиков) ожидают, что состояние потока будет соответствовать стандартным библиотечным значениям. В этих случаях оставленный в нестандартном состоянии поток может привести к ошибке. В результате обычно лучше отменить изменение состояния, как только оно больше не нужно.

Контроль формата логических значений

Хорошим примером манипулятора, изменяющего состояние формата своего объекта, является манипулятор `boolalpha`. По умолчанию значение типа `bool` выводится как 1 или 0. Значение `true` выводится как целое число 1, а значение `false` как 0. Это поведение можно переопределить, применив к потоку манипулятор `boolalpha`:

```
cout << "default bool values: " << true << " "
false
<< "\nalpha bool values: " << boolalpha
<< true << " " << false << endl;
```

Эта программа выводит следующее:

```
default bool values: 1 0
alpha bool values: true false
```

Как только манипулятор `boolalpha` "записан" в поток `cout`, способ вывода логических значений изменяется. Последующие операции вывода логических значений отобразят их как "`true`" или "`false`".

Чтобы отменить изменение флага формата потока `cout`, применяется манипулятор `noboolalpha`:

```
bool bool_val = get_status();
```

```
cout << boolalpha // устанавливает внутреннее
состояние cout
    << bool_val
        << noboolalpha; // возвращает стандартное
внутреннее состояние
```

Здесь формат вывода логических значений изменен только для вывода значения `bool_val`. Как только это значение будет выведено, поток немедленно возвращается в первоначальное состояние.

Определение базы целочисленных значений

По умолчанию целочисленные значения выводятся и читаются в десятичном формате. Используя манипуляторы `hex`, `oct` и `dec`, базу записи числа можно изменить на восьмеричную, шестнадцатеричную и обратно на десятичную базу:

```
cout << "default: " << 20 << " " << 1024 << endl;
cout << "octal: " << oct << 20 << " " << 1024 <<
endl;
cout << "hex: " << hex << 20 << " " << 1024 <<
endl;
cout << "decimal: " << dec << 20 << " " << 1024 <<
endl;
```

После компиляции и запуска на выполнение эта программа выводит следующее:

```
default: 20 1024
octal: 24 2000
hex: 14 400
decimal: 20 1024
```

Обратите внимание, как и манипулятор `boolalpha`, эти манипуляторы изменяют флаг формата. Они срабатывают сразу после применения и влияют на весь последующий вывод целочисленных значений, пока формат не изменит применение другого манипулятора.



Манипуляторы `hex`, `oct` и `dec` влияют на вывод только целочисленных операндов, но не значений с плавающей запятой.

Индикация базы числа в выводе

По умолчанию при выводе числа нет никакого визуального уведомления об используемой базе. Например, 20 — это действительно 20, или восьмеричное представление числа 16? Когда числа выводятся в десятичном режиме, они отображаются, как и ожидается. Если необходимо выводить восьмеричные или шестнадцатеричные значения, вероятней всего, придется использовать также манипулятор `showbase`. Он заставляет поток вывода использовать те же соглашения, что и при определении базы целочисленных констант.

- Предваряющий 0x означает шестнадцатеричный формат.
- Предваряющий 0 означает восьмеричный формат.
- Отсутствие любого индикатора означает десятичное число.

Здесь предыдущая программа пересмотрена для использования манипулятора `showbase`:

```
cout << showbase; // отображать базу при выводе
целочисленных значений
cout << "default: " << 20 << " " << 1024 << endl;
cout << "in octal: " << oct << 20 << " " << 1024 <<
endl;
cout << "in hex: " << hex << 20 << " " << 1024 <<
endl;
cout << "in decimal: " << dec << 20 << " " << 1024
<< endl;
```

cout << noshowbase; // возвратить состояние потока

Вывод пересмотренной программы проясняет смысл:

```
default: 20 1024
in octal: 024 02000
in hex: 0x14 0x400
in decimal: 20 1024
```

Манипулятор `noshowbase` возвращает поток `cout` в прежнее состояние, когда индикатор базы не отображается.

По умолчанию шестнадцатеричные значения выводятся в нижнем регистре с x, также в нижним регистре. Манипулятор `uppercase` позволяет отобразить X и шестнадцатеричные цифры a-f в верхнем регистре:

```
cout << uppercase << showbase << hex
    << "printed in hexadecimal: " << 20 << " " <<
1024
    << nouppercase << noshowbase << dec << endl;
```

Этот оператор создает следующий вывод:

```
printed in hexadecimal: 0X14 0X400
```

Манипуляторы `nouppercase`, `noshowbase` и `dec` применяются для возвращения потока в исходное состояние.

Контроль формата значений с плавающей точкой

Контролировать можно три аспекта вывода числа с плавающей запятой.

- Количество выводимых цифр точности.
- Выводится ли число в шестнадцатеричном формате, как фиксированное десятичное число или в экспоненциальном представлении.
- Выводится ли десятичная точка для целочисленных значений с плавающей запятой.

По умолчанию значения с плавающей запятой выводятся с шестью цифрами точности; десятичная точка не отображается при отсутствии дробной части; в зависимости от величины значения используется фиксированный десятичный формат или экспоненциальная форма. Библиотека выбирает формат, увеличивающий удобочитаемость числа. Очень большие и очень маленькие значения выводятся в экспоненциальном представлении. Другие значения выводятся в фиксированном десятичном формате.

Определение точности

По умолчанию точность контролирует общее количество отображаемых цифр. При выводе значение с плавающей запятой округляется (а не усекается) до текущей точности. Таким образом, если текущая точность четыре, то число 3.14159 становится 3.142; если точность три, то оно выводится как 3.14.

Для изменения точности можно воспользоваться функцией-членом `precision()` объекта ввода-вывода или манипулятором `setprecision`. Функция-член `precision()` перегружена (см. раздел 6.4). Одна ее версия получает значение типа `int` и устанавливает точность в это новое значение. Она возвращает *предыдущее* значение точности. Другая версия не получает никаких аргументов и возвращает текущее значение точности. Манипулятор `setprecision` получает аргумент, который и использует для установки точности.



Манипулятор `setprecision` и другие манипуляторы, получающие аргументы, определяются в заголовке `iomanip`.

Следующая программа иллюстрирует различные способы контроля точности при выводе значения с плавающей точкой:

```
// cout.precision() сообщает текущее значение
точности
cout << "Precision: " << cout.precision()
     << ", Value: " << sqrt(2.0) << endl;
// cout.precision(12) запрашивает вывод 12 цифр
точности
cout.precision(12);
cout << "Precision: " << cout.precision()
     << ", Value: " << sqrt(2.0) << endl;
// альтернативный способ установки точности с
использованием
// манипулятора
setprecision cout << setprecision(3);
cout << "Precision: " << cout.precision()
     << ", Value: " << sqrt(2.0) << endl;
```

Эта программа выводит следующее:

```
Precision: 6, Value: 1.41421
Precision: 12, Value: 1.41421356237
Precision: 3, Value: 1.41
```

Программа использует библиотечную функцию `sqrt()`, определенную в заголовке `cmath`. Функция `sqrt()` перегружена и может быть вызвана с аргументами типа `float`, `double` или `long double`. Она возвращает квадратный корень своего аргумента.

Определение формы записи чисел с плавающей запятой



Рекомендуем

Если нет реальной необходимости контролировать представление числа с плавающей запятой (например, для вывода данных в столбик, отображения денежных данных или процентов), лучше позволить библиотеке выбирать форму записи самостоятельно.

Используя соответствующий манипулятор, можно заставить поток

использовать научную, фиксированную или шестнадцатеричную форму записи. Манипулятор `scientific` задает использование экспоненциального представления. Манипулятор `fixed` задает использование фиксированных десятичных чисел.



Новая библиотека позволяет выводить значения с плавающей точкой в шестнадцатеричном формате при помощи манипулятора `hexfloat`. Новая библиотека предоставляет еще один манипулятор, `defaultfloat`. Он возвращает поток в стандартное состояние, при котором выбор формы записи осуществляется на основании выводимого значения.

Эти манипуляторы изменяют также заданное для потока по умолчанию значение точности. После применения манипуляторов `scientific`, `fixed` или `hexfloat` значение точности контролирует количество цифр после десятичной точки. По умолчанию точность определяет количество цифр до и после десятичной точки. Манипуляторы `fixed` и `scientific` позволяют выводить числа, выстроенные в столбцы, с десятичной точкой в фиксированной позиции относительно дробной части:

```
cout << "default format: " << 100 * sqrt(2.0) <<
' \n'
           << "scientific: " << scientific << 100 *
sqrt(2.0) << ' \n'
           << "fixed decimal: " << fixed << 100 *
sqrt(2.0) << ' \n'
           << "hexadecimal: " << hexfloat << 100 *
sqrt(2.0) << ' \n'
           << "use defaults: " << defaultfloat << 100 *
sqrt(2.0)
           << "\n\n";
```

Получается следующий вывод:

```
default format: 141.421
scientific: 1.414214e+002
fixed decimal: 141.421356
hexadecimal: 0x1.1ad7bcp+7
use defaults: 141.421
```

По умолчанию шестнадцатеричные цифры и символ `e`, используемый в экспоненциальном представлении, выводятся в нижнем регистре. Манипулятор `uppercase` позволяет выводить эти значения в верхнем

регистре.

Вывод десятичной точки

По умолчанию, когда дробная часть значения с плавающей точкой равна 0, десятичная точка не отображается. Манипулятор `showpoint` требует отображать десятичную точку всегда:

```
cout << 10.0 << endl;           // выводит 10
cout << showpoint << 10.0      // выводит 10.0000
                           << noshowpoint << endl; // возвращает
стандартный формат
                           // десятичной точки
```

Манипулятор `noshowpoint` восстанавливает стандартное поведение. У вывода следующих выражений будет стандартное поведение, подразумевающее отсутствие десятичной точки, если дробная часть значения с плавающей точкой отсутствует.

Дополнение вывода

При выводе данных в столбцах зачастую необходим довольно подробный контроль над форматированием данных. Библиотека предоставляет несколько манипуляторов, обеспечивающих контроль, который может понадобиться.

- Манипулятор `setw` задает минимальное пространство для *следующего* числового или строкового значения.
- Манипулятор `left` выравнивает текст по левому краю вывода.
- Манипулятор `right` выравнивает текст по правому краю (принято по умолчанию).
- Манипулятор `internal` контролирует положение знака отрицательных значений. Выравнивает знак по левому краю, а значение по правому, дополняя пространство между ними пробелами.
- Манипулятор `setfill` позволяет задать альтернативный символ для дополнения вывода. По умолчанию принят пробел.



Манипуляторы `setw` и `endl` не изменяют внутреннее состояние потока вывода. Они определяют только *следующий* вывод.

Эти манипуляторы иллюстрирует следующая программа:

```

int i = -16;
double d = 3.14159;
// дополняет первый столбец, обеспечивая минимум 12
позиций вывода
cout << "i: " << setw(12) << i << "next col" <<
'\n'
                << "d: " << setw(12) << d << "next col" <<
'\n';
// дополняет первый столбец и выравнивает все
столбцы по левому краю
cout << left
                << "i: " << setw(12) << i << "next col" <<
'\n'
                << "d: " << setw(12) << d << "next col" <<
'\n'
                << right; // восстанавливает стандартное
выравнивание
// дополняет первый столбец и выравнивают все
столбцы по правому краю
cout << right
                << "i: " << setw(12) << i << "next col" <<
'\n'
                << "d: " << setw(12) << d << "next col" <<
'\n';
// дополняет первый столбец и помещает дополнение в
поле
cout << internal
                << "i: " << setw(12) << i << "next col" <<
'\n'
                << "d: " << setw(12) << d << "next col" <<
'\n';
// дополняет первый столбец, используя символ # как
заполнитель
cout << setfill('#')
                << "i: " << setw(12) << i << "next col" <<
'\n'
                << "d: " << setw(12) << d << "next col" <<
'\n'
                << setfill(' '); // восстанавливает

```

стандартный символ заполнения

Вывод этой программы таков:

```
i:          -16next col
d:          3.14159next col
i: -16      next col
d: 3.14159   next col
i:          -16next col
d:          3.14159next col
i: -       16next col
d: 3.14159next col
i: -#####16next col
d: #####3.14159next col
```

Контроль формата ввода

По умолчанию операторы ввода игнорируют символы отступа (пробел, табуляция, новая строка, новая страница и возврат каретки).

```
char ch;
while (cin >> ch)
    cout << ch;
```

Этот цикл получает следующую исходную последовательность:

**a b c
d**

Он выполняется четыре раза, читая символы от a до d, пропуская промежуточные пробелы, возможные символы табуляции и новой строки. Вывод этой программы таков:

abcd

Манипулятор noskipws заставляет оператор ввода читать, не игнорируя отступ. Для возвращения к стандартному поведению применяется манипулятор skipws:

```
cin >> noskipws; // установить cin на чтение
отступа
while (cin >> ch)
    cout << ch;
cin >> skipws; // возвратить cin к стандартному
игнорированию отступа
```

При том же вводе этот цикл делает семь итераций, читая отступы как символы во вводе. Его вывод таков:

**a b c
d**

Упражнения раздела 17.5.1

Упражнение 17.34. Напишите программу, иллюстрирующую использование каждого манипулятора из табл. 17.17 и 17.18.

Упражнение 17.35. Напишите версию программы вывода квадратного корня, но выводящую на сей раз шестнадцатеричные цифры в верхнем регистре.

Упражнение 17.36. Измените программу из предыдущего упражнения так, чтобы различные значения с плавающей точкой выводились в столбце.

17.5.2. Не форматированные операции ввода-вывода

До сих пор в программах использовались только операции *форматированного ввода-вывода* (formatted IO). Операторы ввода и вывода (<< и >>) форматируют читаемые и выводимые данные согласно их типу. Операторы ввода игнорируют отступ; операторы вывода применяют дополнение, точность и т.д.

Библиотека предоставляет также набор низкоуровневых функций *не форматированного ввода-вывода* (unformatted IO). Эти функции позволяют работать с потоком как с последовательностью неинтерпретируемых байтов.

Однобайтовые операции

Некоторые из не форматированных операций имеют дело с обработкой потока по одному байту за раз. Они описаны в табл. 17.19 и читают данные, не игнорируя отступ. Например, функции не форматированного ввода-вывода `get()` и `put()` позволяют читать и записывать символы по одному:

```
char ch;
while (cin.get(ch))
    cout.put(ch);
```

Эта программа сохраняет отступ во вводе. Ее вывод идентичен вводу. Она работает так же, как и предыдущая программа, использовавшая манипулятор `noskipws`.

Таблица 17.19. Однобайтовые низкоуровневые функции ввода-вывода

<code>is.get(ch)</code>	Помещает следующий байт из потока <code>is</code> класса <code>istream</code> в символьную переменную <code>ch</code> . Возвращает поток <code>is</code>
<code>os.put(ch)</code>	Помещает символ <code>ch</code> в поток <code>os</code> класса <code>ostream</code> . Возвращает поток <code>os</code>
<code>is.get()</code>	Возвращает следующий байт из потока <code>is</code> как тип <code>int</code>
<code>is.putback(ch)</code>	Помещает символ <code>ch</code> назад в поток <code>is</code> ; возвращает поток <code>is</code>
<code>is.unget()</code>	Перемещает в поток <code>is</code> один байт; возвращает поток <code>is</code>
<code>is.peek()</code>	Возвращает следующий байт как тип <code>int</code> , но не удаляет его

Возвращение во входной поток

Иногда необходимо читать отдельные символы так, чтобы знать, к чему быть готовым. В таких случаях символы желательно возвращать в поток. Библиотека предоставляет три способа сделать это, и у каждого из них есть свои отличия.

- Функция `peek()` возвращает копию следующего символа во входном потоке, но не изменяет поток. Возвращенное значение остается в потоке.
- Функция `unget()` создает резервную копию входного потока, чтобы независимо от того, какое значение было последним возвращенным, оно все еще оставалось в потоке. Функцию `unget()` можно вызвать, даже не зная, какое значение было извлечено из потока последним.
- Функция `putback()` — это более специализированная версия функции `unget()`: она возвращает последнее прочитанное из потока значение, но получает аргумент, который должен совпадать с последним прочитанным значением.

Таким образом, они гарантируют возможность вернуть в поток как минимум одно значение перед следующим чтением. Следовательно, гарантированно не получится вызвать функции `putback()` или `unget()` последовательно, без промежуточной операции чтения.

Возвращение значения типа `int` из операций ввода

Функция `peek()` и версия функции `get()` без аргументов возвращают прочитанный символ из входного потока как значение типа `int`. Этот факт может удивить; казалось бы, более естественным было бы возвращение типа `char`.

Причина возвращения этими функциями типа `int` в том, чтобы позволить им возвратить маркер конца файла. Полученный набор символов позволяет использовать каждое значение в диапазоне типа `char` и представлять фактические символы. Но в этом диапазоне нет никакого специального значения для представления конца файла.

Функции, возвращающие тип `int`, преобразуют возвращаемый символ в тип `unsigned char`, а затем преобразуют это значение в тип `int`. В результате, даже если в наборе символов будут символы, соответствующие отрицательным значениям, возвращенный этими функциями тип `int` будет иметь положительное значение (см. раздел 2.1.2). Библиотека использует отрицательное значение для представления конца файла, гарантируя таким образом его отличие от любого настоящего символьного значения. Чтобы не обязывать разработчиков знать фактическое возвращаемое значение, заголовок `iostream` определяет константу `EOF`,

которую можно использовать для проверки, не является ли возвращенное функцией `get()` значение концом файла. Вот почему для содержания значения, возвращаемого этими функциями, используется переменная типа `int`.

```
int ch; // возвращаемое fromget() значение
содержится в int, а не char
// цикл чтения и записи всех данных во вводе
while ((ch = cin.get()) != EOF)
    cout.put(ch);
```

Эта программа работает так же, как и прежняя, но здесь для чтения ввода используется функция `get()`.

Внимание! Низкоуровневые функции подвержены ошибкам

Обычно рекомендуется использовать высокоуровневые абстракции, предоставляемые библиотекой. Функции ввода-вывода, возвращающие значение типа `int`, являются хорошим подтверждением правильности этой рекомендации.

Обычной ошибкой программирования является присвоение значения, возвращаемого функцией `get()` или `peek()`, возвращающей тип `int`, переменной типа `char`, а не `int`. Это, безусловно, будет ошибкой, но компилятор ее не обнаружит. То, что произойдет в результате этой ошибки, зависит от конкретной машины и введенных данных. Например, если машина интерпретирует символ как беззнаковое целое число, приведенный ниже цикл окажется бесконечным.

```
char ch; // применение типа char здесь приведет к
катастрофе!
// значение, возвращенное функцией get() объекта с
in,
// преобразуется из int в char, а затем
сравнивается с int
while ((ch = cin.get()) != EOF)
    cout.put(ch);
```

Проблема в том, что когда функция `get()` возвращает значение `EOF`, оно преобразуется в беззнаковое значение типа `unsigned char`. Это преобразованное значение не будет равно целочисленному значению `EOF`, поэтому цикл не закончится никогда. Такие ошибки обычно обнаруживаются при проверке.

Но нельзя быть уверенными в том, что на тех машинах, где символы

интерпретируются как знаковый топ, поведение цикла будет аналогичным. Ведь результат переполнения переменной беззнакового типа зависит от компилятора. На большинстве машин этот цикл будет работать нормально, если только во вводимых данных не встретится символ, соответствующий значению EOF. Поскольку в обычных данных такие символы маловероятны, низкоуровневые операторы ввода-вывода могут пригодиться при чтении только бинарных значений, которые не соответствуют непосредственно обычным символам и числовым значениям. На машине автора, например, цикл преждевременно завершается в случае ввода символа, значением которого является '\377'. Когда значение '\377' на машине автора преобразуется в тип `signed char`, получается значение -1. Если во введенных данных встретится это значение, оно будет рассматриваться как символ (прежде всего) конца файла.

При чтении и записи типизированных значений такие ошибки не возникают. Поэтому по возможности следует использовать предоставляемые библиотекой высокоДуровневые операторы, что гораздо безопасней.

Многобайтовые операции

Некоторые операции не форматированного ввода-вывода работают с порциями данных за раз. Эти операции могут быть полезны, если важна скорость, но, как и другие низкоуровневые операции, они подвержены ошибкам. В частности эти операции требуют резервирования и управления символьными массивами (см. раздел 12.2), используемыми для сохранения и возвращения данных. Многобайтовые операции перечислены в табл. 17.20.

Таблица 17.20. Многобайтовые низкоуровневые операции ввода-вывода

<code>is.get(sink, size, delim)</code>	Читает до <code>size</code> байтов из потока <code>is</code> и сохраняет их в символьном массиве, начиная с адреса, на который указывает <code>sink</code> . Чтение продолжается, пока не встретится символ <code>delim</code> , либо пока не прочитано <code>size</code> байтов, либо пока не кончится файл. Если параметр <code>delim</code> присутствует, то его значение остается во входном потоке и не читается в <code>sink</code>
<code>is.getline(sink, size, delim)</code>	То же поведение, что и версии функции <code>get()</code> с тремя аргументами, но читает и отбрасывает <code>delim</code>
<code>is.read(sink, size)</code>	Читает до <code>size</code> байтов в символьный массив <code>sink</code> . Возвращает

<code>size)</code>	поток <code>is</code>
<code>is.gcount()</code>	Возвращает количество байтов, прочитанных из потока <code>is</code> при последним вызове функции не форматированного чтения
<code>os.write(source, size)</code>	Записывает <code>size</code> байтов из символьного массива <code>source</code> в поток <code>os</code>
<code>is.ignore(size, delim)</code>	Читает и игнорирует до <code>size</code> символов, включая <code>delim</code> . В отличие от других не форматированных функций, <code>ignore()</code> имеет аргументы по умолчанию: для <code>size</code> — 1 и для <code>delim</code> — конец файла

Функции `get()` и `getline()` имеют схожие, но не идентичные параметры. В каждом случае `sink` — это символьный массив, в который помещаются данные. Обе функции читают, пока не будет выполнено одно из следующих условий:

- Прочитано `size` – 1 символов.
- Встретился конец файла.
- Встретился символ разделения.

Эти функции отличаются обработкой разделителя: функция `get()` оставляет разделитель как следующий символ потока `istream`, а функция `getline()` читает и отбрасывает разделитель. В любом случае разделитель *не сохраняется* в массиве `sink`.



ВНИМАНИЕ

Весьма распространенная ошибка: намереваться удалить разделитель из потока, но забыть сделать это.

Определение количества читаемых символов

Некоторые из операций читают из ввода неизвестное количество байтов. Для определения количества символов, прочитанных последней операцией не форматированного ввода, можно вызвать функцию `gcount()`. Имеет смысл вызывать функцию `gcount()` перед любым вмешательством в операции не форматированного ввода. В частности, операции с единичными символами, возвращающими их в поток, также являются операциями не форматированного ввода. Если функции `peek()`, `unget()` или `putback()` будут вызваны перед вызовом функции `gcount()`, то будет возвращено значение 0.

Упражнения раздела 17.5.2

Упражнение 17.37. Используйте не форматированную версию функции `getline()` для чтения файла по строке за раз. Проверьте программу на примере файла с пустыми строками, а также со строками, длина которых больше символьного массива, переданного функции `getline()`.

Упражнение 17.38. Дополните программу из предыдущего упражнения так, чтобы выводить каждое прочитанное слово в отдельной строке.

17.5.3. Произвольный доступ к потоку

Некоторые из потоковых классов обеспечивают произвольный доступ к данным связанного с ними потока. Положение в потоке можно изменить так, чтобы прочитать сначала последнюю строку, затем первую и т.д. Для *установки* (*seek*) необходимой позиции и *сообщения* (*tell*) текущей позиции в потоке библиотека предоставляет пару функций.



Произвольный доступ для чтения и записи напрямую зависит от системы. Чтобы выяснить способ применения этой возможности, следует обратиться к документации на систему.

Хотя функции `seek()` и `tell()` определены для всех потоковых классов, возможные для них действия определяются видом объекта, с которым связан поток. В большинстве систем поток, с которым связан потоковый объект `cin`, `cout`, `cerr` или `clog`, не обеспечивает возможности произвольного доступа — в конце концов, как можно перейти на десять позиций обратно, если запись осуществляется непосредственно в объект `cout`? Применить функции `seek()` и `tell()`, конечно, можно, но во время выполнения это приведет к ошибке и переходу потока в недопустимое состояние.



Поскольку классы `istream` и `ostream` обычно не обеспечивают произвольного доступа, в остальной части этого раздела речь идет только о классах `fstream` и `sstream`.

Функции установки и сообщения

Для обеспечения произвольного доступа типы ввода-вывода обладают *маркером* (*marker*), который указывает позицию следующей операции чтения или записи. Они обладают также двумя функциями: одна *устанавливает* (*seek*) маркер в новую позицию, а вторая *сообщает* (*tell*) текущую позицию маркера. Фактически в библиотеке определены две пары

функций установки и сообщения, которые описаны в табл. 17.21. Одна пара функций используется потоками ввода, а вторая — потоками вывода. Версии для ввода и вывода различаются суффиксом. Суффикс *g* (getting) означает получение данных (чтение), а суффикс *p* (putting) — помещение данных (запись).

Таблица 17.21. Функции установки и сообщения

<code>tellg()</code> <code>tellp()</code>	Возвращает текущую позицию маркера потока ввода (<code>tellg()</code>) или потока вывода (<code>tellp()</code>)
<code>seekg(pos)</code> <code>seekp(pos)</code>	Переустанавливает маркер потока ввода или вывода на заданный параметром <i>pos</i> абсолютный адрес в потоке. Значение <i>pos</i> обычно возвращается предыдущим вызовом в соответствующей функции <code>tellg()</code> или <code>tellp()</code>
<code>seekp(off, from)</code> <code>seekg(off, from)</code>	Переустанавливает маркер потока ввода или вывода на <i>off</i> символов вперед или назад от значения <i>from</i> , которое может быть: beg — от начала потока;
	cur — от текущей позиции потока;
	end — от конца потока

Вполне логично, что для класса `istream`, а также производных от него классов `ifstream` и `istringstream` (см. раздел 8.1) можно использовать только версии *g*, а для классов `ostream` и классов `ofstream` и `ostringstream`, производных от него, можно использовать только версии *p*. Классы `iostream`, `fstream` и `stringstream` способны читать и записывать данные в поток, поэтому для них можно использовать обе версии, *g* и *p*.

Существует только один маркер

Тот факт, что библиотека различает версии функций `seek()` и `tell()` для чтения и записи, может ввести в заблуждение. Хотя библиотека и различает эти функции, в файле существует только один маркер, т.е. нет разных маркеров для чтения и записи.

Когда речь идет о потоке только ввода или вывода, различие не столь очевидно. В таких потоках можно использовать версии только *g* или *p*. Если попытаться вызвать функцию `tellp()` для объекта класса `ifstream`, компилятор сообщит об ошибке. Аналогично он поступит при попытке вызвать функцию `seekg()` для объекта класса `ostringstream`.

Типы `fstream` и `stringstream` допускают чтение и запись в тот же

поток. У них есть один буфер для хранения подлежащих чтению и записи данных, а также один маркер, обозначающий текущую позицию в буфере. Библиотечные функции версий `g` и `p` используют тот же маркер позиции.



Поскольку существует только один маркер, для переустановки маркера при каждом переключении между чтением и записью *следует* применять функцию `seek()`.

Перемещение маркера

Имеются две версии функции установки позиции: одна обеспечивает переход к указанной позиции в файле, а другая осуществляет смещение от текущей позиции.

```
// установка маркера в заданную позицию
seekg( new_position); // установить маркер чтения в
позицию pos_type
seekp( new_position); // установить маркер записи в
позицию pos_type

// смещение позиции на указанную дистанцию от
текущей
seekg( offset, from); // установить дистанцию
смещения маркера чтения
seekp( offset, from); // от from; offset имеет тип
off_type
```

Возможные значения параметра `from` перечислены в табл. 17.21.

Аргументы `new_position` и `offset` этих функций имеют машинно-зависимые типы `pos_type` и `off_type` соответственно. Они определены в классах `istream` и `ostream`. Тип `pos_type` представляет позицию файла, а тип `off_type` — смещение от этой позиции. Значение типа `off_type` может быть положительным или отрицательным, что соответствует смещению вперед или назад.

Доступ к маркеру

Функции `tellg()` и `tellp()` возвращают значение типа `pos_type`, обозначающее текущую позицию в потоке. Эти функции обычно

используются для того, чтобы запомнить позицию и впоследствии вернуться к ней:

```
// запомнить текущую позицию записи в переменную  
mark
```

```
ostringstream writeStr; // поток вывода в строку  
ostringstream::pos_type mark = writeStr.tellp();  
// ...  
if (cancelEntry)  
    // возврат к отмеченной позиции  
    writeStr.seekp( mark );
```

Чтение и запись в том же файл

Рассмотрим пример программы, которая читает файл и записывает в его конец новую строку, содержащую относительную позицию начала каждой строки. Предположим, например, что работать придется со следующим файлом.

```
Abcd  
efg  
hi  
j
```

Модифицированный программой файл должен выглядеть следующим образом.

```
Abcd  
efg  
hi  
j  
5 9 12 14
```

Обратите внимание: программа не записывает смещение для первой строки, она всегда начинается с позиции 0. Обратите также внимание на то, что смещения должны также учитывать невидимый символ новой строки, завершающий каждую строку. И наконец, последнее число в выводе — смещение для строки, с которой начинается вывод. При включении этого смещения в вывод можно отличить свой вывод от первоначального содержимого файла. Можно прочитать последнее число в полученном файле и установить смещение так, чтобы получить позицию начала вывода.

Наша программа будет читать файл построчно. Для каждой строки значение счетчика будет увеличиваться на размер только что прочитанной строки. Этот счетчик содержит смещение, с которого начинается

следующая строка:

```
int main() {
    // открыть файл для ввода и вывода, а затем
перейти в его конец
    // аргументы режима файла приведены в табл. 8.4
fstream inOut("copyOut",
    fstream::ate | fstream::in | fstream::out);
if (!inOut) {
    cerr << "Unable to open file!" << endl;
    return EXIT_FAILURE; // EXIT_FAILURE см. р. 6.3.2
}
// inOut открыт в режиме ate, поэтому исходной
позицией файла будет
// его конец
auto end_mark = inOut.tellg(); // запомнить
позицию первоначального
// конца файла
inOut.seekg(0, fstream::beg); // перейти к началу
файла
size_t cnt = 0; // счетчик
количество байтов
string line; // содержит каждую
строку ввода
// пока нет ошибки и исходные данные читаются
while (inOut && inOut.tellg() != end_mark
        && getline(inOut, line)) { // и можно
получить следующую строку
    cnt += line.size() + 1; // добавить 1
для новой строки
    auto mark = inOut.tellg(); // запомнить
позицию чтения
    inOut.seekp(0, fstream::end); // установить
маркер записи в конец
    inOut << cnt; // записать
общую длину
    // вывести разделитель, если это не последняя
строка
    if (mark != end_mark) inOut << " ";
    inOut.seekg(mark); // восстановить позицию чтения
```

```

    }
    inOut.seekp(0, fstream::end); // перейти к концу
    inOut << "\n"; // вывести символ новой строки в
конце файла
    return 0;
}

```

Эта программа открывает поток `fstream` в режимах `in`, `out` и `ate` (см. табл. 8.4). Первые два режима означают, что предполагается чтение и запись в тот же файл. Режим `ate` означает, что начальной позицией открытого файла будет его конец. Как обычно, необходимо удостовериться, что файл открыт корректно, если это не так, следует выйти из программы (см. раздел 6.3.2).

Поскольку программа пишет в свой исходный файл, нельзя использовать конец файла как признак прекращения чтения. Цикл должен закончиться по достижении конца первоначального ввода. В результате сначала следует запомнить первоначальную позицию конца файла. Так как файл открыт в режиме `ate`, поток `inOut` уже установлен в конец. Сохраним текущую (т.е. первоначальную) позицию конца файла в переменной `end_mark`. Запомнив конечную позицию, маркер чтения следует установить в начало файла, чтобы можно было приступить к чтению данных.

Цикл `while` имеет три условия выхода: сначала проверяется допустимость потока; если это так, то проверяется, не достигнут ли конец исходных данных. Для этого текущая позиция чтения, возвращаемая функцией `tellg()`, сравнивается с позицией, заранее сохраненной в переменной `end_mark`. И наконец, если обе проверки пройдены успешно, происходит вызов функции `getline()`, которая читает следующую строку из файла. Если вызов функции `getline()` успешен, выполняется тело цикла.

Тело цикла начинается с запоминания текущей позиции в переменной `mark`. Она сохраняется для возвращения после записи следующего относительного смещения. Вызов функции `seekp()` переводит маркер записи в конец файла. Выводится значение счетчика, а затем функция `seekg()` возвращается к позиции, сохраненной в переменной `mark`. Восстановив положение маркера, можно снова проверить условие выхода из цикла `while`.

Каждая итерация цикла выводит смещение следующей строки. Поэтому последняя итерация цикла заботится о записи смещения последней строки.

Однако в конец файла следует еще записать символ новой строки. Как и в других случаях записи, для позиционирования в конец файла перед выводом новой строки происходит вызов функции `seekp()`.

Упражнения раздела 17.5.3

Упражнение 17.39. Напишите собственную версию программы, представленной в этом разделе.

Резюме

В этой главе рассматривались дополнительные операции ввода-вывода и четыре библиотечных типа: кортеж, набор битов, регулярные выражения и случайные числа.

Шаблон `tuple` (кортеж) позволяет объединять члены несопоставимых типов в единый объект. Каждый кортеж содержит конкретное количество членов, но библиотека не налагает ограничений на их количество.

Тип `bitset` (набор битов) позволяет определять коллекции битов определенного размера. Размер набора битов не ограничен размером любого из целочисленных типов и вполне может превышать их. Кроме поддержки обычных побитовых операторов (см. раздел 4.8), набор битов определяет несколько специальных операторов, которые позволяют манипулировать состоянием отдельных битов в наборе.

Библиотека регулярных выражений предоставляет коллекцию классов и функций: класс `regex` представляет регулярные выражения, написанные на одном из нескольких общепринятых языков регулярных выражений. Классы соответствия содержат информацию о конкретном соответствии. Они используются функциями `regex_search()` и `regex_match()`. Эти функции получают объект класса `regex` и последовательность символов, а затем обнаруживают соответствия регулярного выражения `regex` в данной последовательности символов. Итераторы типа `regex` являются адаптерами итераторов, используемых функцией `regex_search()` для перебора исходной последовательности и возвращения каждого соответствия. Есть также функция `regex_replace()`, позволяющая заменять соответствующие части заданной исходной последовательности указанной альтернативой.

Библиотека случайных чисел — это коллекция процессоров случайных чисел и классов распределения. Процессор случайных чисел возвращает последовательность равномерно распределенных целочисленных значений. Библиотека определяет несколько процессоров с разной

производительностью. Процессор `default_random_engine` определен как подходящий для большинства случаев. Библиотека определяет также 20 типов распределений. Эти типы распределений используют процессор как источник случайных чисел определенного типа в заданном диапазоне, которые распределены согласно заданной вероятности распределения.

Термины

Генератор случайных чисел (random-number generator). Комбинация типа процессора случайных чисел и типа распределения.

Исключение`regex_error`. Тип исключения, передаваемого при синтаксической ошибке в регулярном выражении.

Итератор`sregex_iterator`. Подобен итератору `sregex_iterator`, но перебирает массив типа `char`.

Итератор`sregex_iterator`. Итератор, перебирающий строку с использованием заданного объекта класса `regex` для поиска соответствий в данной строке. При вызове функции `regex_search()` конструктор позиционирует итератор на первое соответствие. Приращение итератора вызывает функцию `regex_search()`, начиная сразу после текущего соответствия в данной строке. Обращение к значению итератора возвращает объект класса `smatch`, описывающий текущее соответствие.

Класс`bitset` (набор битов). Определенный в стандартной библиотеке класс, объект которого содержит коллекцию битов, размер которой известен на момент компиляции, и позволяет выполнять с ним операции по проверке и установке значений.

Класс`cmatch`. Контейнер объектов типа `csub_match`, предоставляющий информацию о соответствии классу `regex` в исходной последовательности типа `const char*`. Первый элемент в контейнере описывает общие результаты поиска соответствия. Последующие элементы описывают результаты для подвыражений.

Класс`regex`. Класс, обслуживающий регулярное выражение.

Класс`smatch`. Контейнер объектов типа `csub_match`, предоставляющий информацию о соответствии классу `regex` в исходной последовательности типа `string`. Первый элемент в контейнере описывает общие результаты поиска соответствия. Последующие элементы описывают результаты для подвыражений.

Манипулятор (manipulator). Подобный функции объект, "манипулирующий" потоком. Манипуляторы применяются как правый

операнд на перегруженные операторы ввода-вывода, << и >>. Большинство манипуляторов изменяет внутреннее состояние объекта. Они зачастую предоставляются парами: один изменяет состояние потока, а второй возвращает поток в стандартное состояние.

Младшие биты (low-order). Биты набора, обладающие самыми маленькими индексами.

Начальное число (seed). Значение, предоставляемое процессору случайных чисел, чтобы перейти к новому пункту в последовательности создаваемых чисел.

Не форматированный ввод-вывод (unformatted IO). Операции, рассматривающие поток как недифференцированный поток байтов. Неформатированные операции налагают все обязанности по управлению вводом и выводом на пользователя.

Подвыражение (subexpression). Заключенный в скобки компонент схемы регулярного выражения.

Процессор случайных чисел (random-number engine). Библиотечный тип, позволяющий создавать беззнаковые случайные числа. Процессоры предназначены для использования только как источники для распределения случайных чисел.

Распределение случайных чисел (random-number distribution). Тип стандартной библиотеки, преобразующий вывод процессора случайного числа согласно его именованному распределению. Например, шаблон `uniform_int_distribution<T>` создает однородно распределенные целые числа типа `T`, шаблон `normal_distribution<T>` создает числа с нормальным распределением и т.д.

Регулярное выражение (regular expression). Способ описания последовательности символов.

Стандартный процессор случайных чисел (default random engine). Псевдоним типа для процессора случайных чисел, предназначенный для обычного использования.

Старшие биты (high-order). Биты набора, обладающие самыми большими индексами.

Тип `sub_match`. Тип, содержащий результаты поиска соответствия регулярного выражения для типа `const char*`. Может представлять все соответствия или подвыражение.

Тип `ssub_match`. Тип, содержащий результаты поиска соответствия регулярного выражения для типа `string`. Может представлять все соответствия или подвыражение.

Форматированный ввод-вывод (formatted IO). Операции ввода-вывода, использующие для определения действий операций типы читаемых или записываемых объектов. Поскольку сложные операции ввода выполняют все соответствующие читаемому типу преобразования, такие как преобразование числовых строк ASCII в указанный арифметический тип, отступ (по умолчанию) игнорируется. Процедуры форматированного вывода преобразуют типы в представления отображаемых символов, дополняя (возможно) вывод и выполняя другие, специфические для типа преобразования.

Функция`regex_match()`. Функция, сообщающая, соответствует ли вся исходная последовательность заданному объекту класса `regex`.

Функция`regex_replace()`. Функция, использующая объект класса `regex` для замены соответствующего подвыражения исходной последовательности с использованием заданного формата.

Функция`regex_search()`. Функция, использующая объект класса `regex` для поиска последовательности соответствия в заданной исходной последовательности.

Шаблон`tuple` (кортеж). Шаблон, позволяющий создавать типы для хранения безымянных членов определенных типов. Нет никаких ограничений на количество членов, для содержания которых может быть определен кортеж.

Шаблон функции`get`. Шаблон функции, возвращающий определенный элемент для заданного кортежа. Например, функция `get<0>(t)`, возвращает первый элемент из кортежа `tuple t`.

Глава 18

Инструменты для крупномасштабных программ

Язык C++ используется для решения проблем любой сложности — как незначительных, которые способен решить один программист за несколько часов вечером после основной работы, так и чудовищно сложных, требующих десятков миллионов строк кода и модифицируемых впоследствии на протяжении многих лет. Средства, описанные в предыдущих разделах этой книги, полезны для решения весьма широкого диапазона вопросов программирования.

Язык предоставляет некоторые средства, которые полезней в больших и сложных системах, чем в простых. Это средства обработки исключений, пространства имен и множественное наследование, являющиеся темой данной главы.

Крупномасштабное программирование предъявляет к языку более высокие требования чем те, которых достаточно для небольших групп разработчиков. К этим требованиям относятся следующие.

- Способность обрабатывать ошибки при помощи независимой подсистемы.
- Способность использовать библиотеки, разработанные более или менее независимо.
- Способность моделировать более сложные прикладные концепции.

В данной главе рассматриваются три предназначенных для этого средства языка C++: обработка исключений, пространства имен и множественное наследование.

18.1. Обработка исключений

Обработка исключений (exception handling) позволяет независимо разработанным частям программы взаимодействовать и решать проблемы, возникающие во время выполнения. Исключения позволяют отделять код обнаружения проблемы от кода ее решения. Часть программы, ответственная за обнаружение проблемы, может передать информацию о возникшей ситуации другой части программы, которая специально предназначена для решения подобных проблем.

Основы концепции применения исключений в языке C++ представлены в разделе 5.6. В данном разделе эта тема рассматривается подробней. Эффективное использование обработки исключений требует понимания происходящего при передаче исключения, его обработки и смысла объектов, сообщающих о том, что пошло не так.

18.1.1. Передача исключений

В языке C++ исключение *передается* (*raise*) выражением `throw` (передача исключения). Тип выражения `throw`, вместе с текущей цепочкой вызова, определяет, какой *обработчик* (*handler*) будет обрабатывать исключение. Выбирается ближайший обработчик в цепочке вызовов, соответствующий типу переданного объекта. Тип и содержимое этого объекта позволяют передающей части программы сообщать обрабатывающей части о том, что пошло не так.

Когда выполняется оператор `throw`, расположенные после него выражения игнорируются. Оператор `throw` передает управление соответствующему блоку `catch`. Блок `catch` может быть локальным для той же функции или функции, непосредственно или косвенно вызвавшей ту, в которой произошла ошибка, приведшая к передаче исключения. Тот факт, что управление передается из одного места в другое, имеет два важных следствия.

- Функции можно преждевременно покидать по цепочке вызовов.
- По достижении обработчика созданные цепочкой вызова объекты будут уничтожены.

Поскольку операторы после оператора `throw` не выполняются, он похож на оператор `return`: он обычно является частью условного оператора или последним (или единственным) оператором функции.

Прокрутка стека

При передаче исключения выполнение текущей функции приостанавливается и начинается поиск соответствующей директивы `catch`. Поиск начинается с проверки того, расположен ли оператор `throw` непосредственно в блоке `try` (`try block`). Если это так, проверяется соответствие переданного объекта одному из обработчиков того блока `catch`, с которым связан данный блок `try`. Если соответствие в блоке `catch` найдено, исключение обрабатывается. В противном случае осуществляется выход из текущей функции, ее память освобождается, а локальные объекты удаляются. Затем поиск продолжается в вызывающей функции.

Если обращение к передавшей исключение функции находится в блоке `try`, проверяются обработчики того блока `catch`, который связан с ним. Если соответствие найдено, исключение обрабатывается. В противном случае осуществляется выход и из вызывающей функции, а поиск продолжается в той функции, которая вызвала ее, и так далее.

Этот процесс, известный как *прокрутка стека* (*stack unwinding*), продолжается по цепи обращений вложенных функций до тех пор, пока не будет найден соответствующий исключению обработчик `catch`, а если он найден не будет, то до конца функции `main()`.

Как только способный обрабатывать исключение блок `catch` будет найден, выполнение продолжится в этом обработчике. По завершении работы обработчика выполнение продолжится с точки, расположенной непосредственно после последней директивы блока `catch`.

Если соответствующий блок `catch` не найден, программа завершает работу. Исключения предназначены для событий, препятствующих нормальному продолжению выполнения программы. Поэтому переданное исключение не может остаться необработанным. Если соответствующий блок `catch` не найден, программа вызывает библиотечную функцию `terminate()`, которая прекращает выполнение программы.



Необработанное исключение завершает программу.

Объекты автоматически удаляются при прокрутке стека

В ходе прокрутки стека происходит преждевременный выход из функции, содержащей оператор `throw`, а возможно, и из других функций

по цепи обращений. Как правило, функции создают локальные объекты, которые при выходе из функции удаляются. При выходе из функции в связи с передачей исключения компилятор гарантирует правильное удаление локальных объектов. Когда завершается работа любой функции, ее локальное хранилище освобождается. Перед освобождением памяти удаляются все локальные объекты, которые были созданы до передачи исключения. Если локальный объект имеет тип класса, для него автоматически вызывается деструктор. Как обычно, для удаления объектов встроенного типа компилятор ничего не делает.

Если исключение происходит в конструкторе, значит, объект находится еще на стадии создания и может быть закончен только частично. Некоторые из его членов, возможно, уже инициализированы, а другие, возможно, нет. Даже если объект создан только частично, следует гарантировать корректное удаление составляющих его членов.

Точно так же исключение могло бы произойти во время инициализации элементов массива или контейнера библиотечного типа. Корректное удаление элементов, созданных прежде, чем произошло исключение, также следует гарантировать.

Деструкторы и исключения

Тот факт, что деструктор запущен, но код в функции, освобождающий ресурс, может быть пропущен, влияет на структуру создаваемых программ. Как упоминалось в разделе 12.1.4, если блок резервирует ресурс, а исключение происходит перед кодом, который его освобождает, освобождающий ресурс код не будет выполнен. С другой стороны, ресурсы, распределенные объектом класса, обычно освобождаются их деструктором. Использование классов для контроля резервирования ресурсов гарантирует правильность их освобождения, если функция завершается нормально или в результате исключения.

Факт запуска деструктора во время прокрутки стека влияет на то, как следует создавать деструкторы. Во время прокрутки стека исключение уже передано, но еще не обработано. Если во время прокрутки стека передается новое исключение и не обрабатывается в передавшей его функции, то вызывается функция `terminate()`. Поскольку деструкторы могут быть вызваны во время прокрутки стека, они никогда не должны передавать исключений, которые не обрабатывает сам деструктор. Таким образом, если деструктор выполняет операцию, которая могла бы передать исключение, он должен заключить ее в блок `try` и обработать локально в деструкторе.

На практике, поскольку деструкторы освобождают ресурсы, маловероятно, что они передадут исключения. Все типы стандартной библиотеки гарантируют, что их деструкторы не будут передавать исключение.



Во время прокрутки стека для локальных объектов классов выполняются деструкторы. Поскольку деструкторы выполняются автоматически, они не должны передавать исключений. Если во время прокрутки стека деструктор передаст исключение, которое он не обрабатывает, то программа будет завершена.

Объект исключения

Компилятор использует выражения передачи исключения для инициализации копией (см. раздел 13.1.1) специального объекта известного как *объект исключения* (exception object). В результате, у выражения в блоке `throw` должен быть полный тип (см. раздел 7.3.3). Кроме того, если у выражения тип класса, то его деструктор, конструктор копий и конструктор перемещения должны быть доступны. Если выражение имеет тип массива или функции, выражение преобразовывается в соответствующий ему тип указателя.

Объект исключения располагается в управляемой компилятором области памяти, которая будет гарантировано доступна для любого обработчика. Объект исключения удаляется после того, как исключение будет полностью обработано.

Как уже упоминалось, при передаче исключения осуществляется выход из всех блоков по цепочке вызовов, пока не будет найден соответствующий обработчик. При выходе из блока вся память, используемая его локальными объектами, освобождается. В результате передача указателя на локальный объект почти наверняка будет ошибкой. Причина этой ошибки та же, что и у ошибки возвращения из функции указателя на локальный объект (см. раздел 6.3.2). Если указатель указывает на объект в блоке, выход из которого осуществляется перед обработчиком, то этот локальный объект будет удален до обработчика.

При передаче исключения его выражение определяет статический тип (тип времени компиляции) (см. раздел 15.2.3) объекта исключения. Этот момент важно иметь в виду, поскольку большинство приложений передают

исключения, тип которых исходит из иерархии наследования. Если выражение `throw` обращается к значению указателя на тип базового класса и этот указатель указывает на объект производного класса, то переданный объект отсекается (см. раздел 15.2.3) и передается только часть базового класса.



Передача указателя требует, чтобы объект, на который указывает указатель, существовал на момент выполнения соответствующего обработчика.

Упражнения раздела 18.1.1

Упражнение 18.1. Каков тип объекта исключения в следующих операторах `throw`?

- (a) `range_error r("error");` (b) `exception *p = &r;`
`throw r;` `throw *p;`

Что было бы, будь оператор `throw` в случае (b) написан как `throw p`?

Упражнение 18.2. Объясните, что случится, если исключение произойдет в указанном месте:

```
void exercise( int *b, int *e) {  
    vector<int> v( b, e);  
    int *p = new int[ v.size() ] ;  
    ifstream in( "ints" );  
    // исключение происходит здесь  
}
```

Упражнение 18.3. Существуют два способа исправить предыдущий код. Опишите и реализуйте их.

18.1.2. Обработка исключения

Обявление исключения (exception declaration) в *директиве catch* (catch clause) выглядит как список параметров функции, только с одним параметром. Как и в списке параметров, имя параметра обработчика можно пропустить, если у блока `catch` нет необходимости в доступе к переданному исключению.

Тип объявления определяет виды исключений, обрабатываемых обработчиком. Тип должен быть завершенным (см. раздел 7.3.3). Тип

может быть ссылкой на l-значение, но не ссылкой на r-значение (см. раздел 13.6.1).

При входе в блок `catch` параметр в объявлении исключения инициализируется объектом исключения. Подобно параметру функции, если тип параметра обработчика не является ссылочным, параметр обработчика копирует объект исключения; изменения, внесенные в параметр в обработчике, осуществляются с его локальной копией, а не с самим объектом исключения. Если параметр имеет ссылочный тип, то, как любой ссылочный параметр, параметр обработчика будет только другим именем объекта исключения. Изменения, внесенные в ссылочный параметр, осуществляются с самим объектом исключения.

Подобно объявлению параметра функции, параметр обработчика, имеющий тип базового класса, может быть инициализирован объектом исключения типа производного класса. Если у параметра обработчика будет не ссылочный тип, то объект исключения будет отсечен (см. раздел 15.2.3), как и при передаче такого объекта обычной функции по значению. С другой стороны, если параметр является ссылкой на тип базового класса, то параметр будет связан с объектом исключения обычным способом.

Также, подобно параметрам функции, статический тип объявления исключения определяет действия, которые может выполнить обработчик. Если у параметра обработчика будет тип базового класса, то обработчик не сможет использовать члены, определенные в производном классе.

Рекомендуем

Обычно обработчики, получающие исключения типа, связанных наследственными отношениями, определяют свой параметр как ссылку.

Поиск соответствующего обработчика

Блок `catch`, найденный в ходе поиска соответствующего обработчика, не обязательно является наиболее подходящим данному исключению. В результате исключение будет обработано *первым* найденным блоком `catch`, который сможет это сделать. Как следствие, в списке директив `catch` наиболее специализированные обработчики следует располагать в начале.

Поскольку поиск директивы `catch` осуществляется в порядке их объявления, при использовании исключений из иерархии наследования блоки `catch` для обработки исключений производного типа следует

располагать перед обработчиком для исключения базового типа.

Правила поиска соответствующего исключению блока `catch` значительно жестче, чем правила поиска аргументов, соответствующих типам параметров. Большинство преобразований здесь недопустимо — тип исключения должен точно соответствовать обработчику, допустимо лишь несколько различий.

- Допустимо преобразование из неконстантного типа в константный, т.е. переданный неконстантный объект исключения может быть обработан блоком `catch`, ожидающим ссылку на константный.
- Допустимо преобразование из производного типа в базовый.
- Массив преобразуется в указатель на тип массива; функция преобразуется в соответствующий указатель на тип функции.

Никакие другие преобразования при поиске соответствующего обработчика недопустимы. В частности, невозможны ни стандартные арифметические преобразования, ни преобразования, определенные для классов.



В наборе директив `catch` с типами, связанными наследованием, обработчики для более производных типов следует располагать прежде наименее производных.

Повторная передача исключения

Вполне возможна ситуация, когда один блок кода `catch` (обработчик) не сможет полностью обработать исключение. После некоторых корректирующих действий обработчик может решать, что это исключение следует обработать в функции, которая расположена далее по цепи вызовов. Обработчик может передавать исключение другому, внешнему обработчику, который принадлежит функции, вызвавшей данную. Это называется *повторной передачей исключения* (*rethrow*). Повторную передачу осуществляет оператор `throw`, после которого нет ни имени типа, ни выражения.

```
throw;
```

Пустой оператор `throw` может присутствовать только в обработчике или в функции, вызов которой осуществляется из обработчика (прямо или косвенно). Если пустой оператор `throw` встретится вне обработчика, будет

вызвана функция `terminate()`.

Повторная передача не определяет нового исключения; по цепочке передается текущий объект исключения.

Обычно обработчик вполне может изменить содержимое своего параметра. Если после изменения своего параметра обработчик повторно передаст исключение, то эти изменения будут переданы далее, только если параметр обработчика объявлен как ссылка:

```
catch ( my_error &eObj) { // спецификатор ссылочного
    // типа
    eObj.status = errCodes::severeErr; // изменение
    // объекта исключения
    throw; // переменная-член status объекта
    // исключения имеет
    // значение severeErr
} catch ( other_error eObj) { // спецификатор
    // нессылочного типа
    eObj.status = errCodes::badErr; // изменение
    // только локальной копии
    throw; // значение переменной-члена status объекта
    // исключения
    // при повторной передаче не изменилось
}
```

Обработчик для всех исключений

Иногда необходимо обрабатывать все исключения, которые могут произойти, независимо от их типа. Обработка каждого возможного исключения может быть проблематична: иногда неизвестно, исключения каких типов могут быть переданы. Даже когда все возможные типы известны, предоставление отдельной директивы `catch` для каждого возможного исключения может оказаться весьма утомительным. Для обработки всех исключений в объявлении исключения используется многоточие. Такие обработчики, называемые *обработчиками для всех исключений* (*catch-all*), имеют форму `catch(...)`. Такая директива соответствует исключениям любого типа.

Обработчик `catch(...)` зачастую используется в комбинации с выражением повторной передачи. Обработчик осуществляет все локальные действия, а затем повторно передает исключение:

```
void manip() {
    try {
```

```
// действия, приводящие к передаче исключения
} catch (...) {
    // действия по частичной обработке исключения
    throw;
}
```

Директива `catch(...)` применяется самостоятельно или в составе нескольких директив `catch`.



Если директива `catch(...)` используется в комбинации с другими, она должна располагаться последней. Любой обработчик, следующий за обработчиком для всех исключений, никогда не будет выполнен.

Упражнения раздела 18.1.2

Упражнение 18.4. Заглянув вперед в иерархию наследования на рис. 18.1, объясните, что неправильно в следующем блоке `try`. Исправьте его:

```
try {
    // использовать стандартную библиотеку C++
} catch( exception ) {
    // ...
} catch( const runtime_error &re) {
    // ...
} catch( overflow_error eobj) { /* ... */ }
```

Упражнение 18.5. Измените следующую функцию `main()` так, чтобы обрабатывались исключения любых типов, представленных на рис. 18.1:

```
int main() {
    // использовать стандартную библиотеку C++
}
```

Обработчики должны выводить сообщения об ошибках, связанных с исключением, прежде, чем вызывать функцию `abort()` (определенную в заголовке `cstdlib`) для завершения функции `main()`.

Упражнение 18.6. С учетом следующих типов исключений и директивы `catch` напишите выражение `throw`, создающее объект исключения, который может быть обработан каждым блоком `catch`:

(а) class exceptionType { };

```
    catch ( exceptionType *pet) { }
(b) catch (...) { }
(c) typedef int EXCPTYPE;
    catch ( EXCPTYPE) { }
```

18.1.3. Блок `try` функции и конструкторы

В принципе исключения могут произойти в любой точке программы. В частности, исключение может произойти в процессе инициализации в конструкторе. Инициализация в конструкторе выполняется прежде, чем его тело. Блок `catch` в теле конструктора не может обработать исключение, которое было передано при инициализации, поскольку блок `try` в теле конструктора еще не был задействован в момент передачи исключения.

Для обработки исключения, переданного при инициализации, конструктор следует оформить как *блок try функции* (function `try` block). Блок `try` функции позволяет ассоциировать группу директив `catch` с фазой инициализации конструктора (или фазой удаления деструктора), а равно с телом конструктора (или деструктора). В качестве примера заключим конструктор `Blob()` (см. раздел 16.1.2) в блок `try` функции:

```
template <typename T>
Blob<T>::Blob( std::initializer_list<T> il) try :
    data( std::make_shared<std::vector<T>>( il)) {
    /* пустое тело */
}      catch( const      std::bad_alloc      &e)      {
handle_out_of_memory( e); }
```

Обратите внимание на ключевое слово `try`, предшествующее двоеточию, начинаящему список инициализации конструктора, и фигурную скобку, формирующую (в данном случае пустое) тело конструктора. Обработчик, связанный с этим блоком `try`, применяется для обработки исключения, переданного либо из списка инициализации, либо из тела конструктора.

Следует заметить, что исключение может произойти при инициализации параметров конструктора. Такие исключения не являются частью блока `try` функции. Блок `try` функции обрабатывает только те исключения, которые происходят, когда конструктор начнет выполняться. Как и при любом другом вызове функции, если исключение происходит во время инициализации параметра, оно является частью вызывающего выражения и обрабатывается в контексте вызывающей стороны.



Единственный способ для конструктора обработать исключение из списка инициализации заключается в оформлении конструктора как блока `try` функции.

Упражнения раздела 18.1.3

Упражнение 18.7. Определите классы `Blob` и `BlobPtr` из главы 16 так, чтобы для их конструкторов использовались блоки `try` функции.

18.1.4. Спецификатор исключения `noexcept`

И для пользователей, и для компилятора может быть полезно знать, что функция не будет передавать исключения. Это упрощает написание кода, вызывающего эту функцию. Кроме того, если компилятор знает, что никаких исключений не будет, он может (иногда) оптимизировать код, что недоступно при возможности передачи.



По новому стандарту функция может пообещать не передавать исключения при помощи *спецификации noexcept*. Ключевое слово `noexcept` после списка параметров функции означает, что функция не будет передавать исключений:

```
void recoup( int ) noexcept; // не будет передавать
исключений
void alloc( int );           // может передавать
исключения
```

Эти объявления заявляют, что функция `recoup()` не будет передавать исключений, а функция `alloc()` могла бы. Считается, что к функции `recoup()` применена *спецификация запрета передачи исключения* (*nonthrowing specification*).

Спецификатор `noexcept` должен присутствовать во всех объявлениях и в соответствующем определении функции или ни в одном из них. Спецификатор предшествует замыкающему типу (см. раздел 6.3.3). Спецификатор `noexcept` можно определить также в объявлении и определении указателя на функцию. Он неприменим к псевдониму типа

или определению типа (`typedef`). В функции-члене спецификатор `noexcept` следует за квалификатором `const` или квалификатором ссылки, но предшествует квалификаторам `final`, `override` и `= 0` у виртуальной функции.

Нарушение спецификации исключения

Важно понимать, что компилятор не проверяет спецификацию `noexcept` во время компиляции. Фактически компилятору не разрешено отклонять функцию со спецификатором `noexcept` просто потому, что она содержит оператор `throw` или вызывает функцию, которая может передавать исключение (однако хорошие компиляторы предупреждают о таких случаях):

```
// эта функция компилируется, хоть она и нарушает
свою спецификацию
// исключения
void f() noexcept      // обещание не передавать
исключений
{
    throw exception(); // нарушает спецификацию
исключений
}
```

В результате вполне вероятно, что функция, обещавшая не передавать исключений, фактически передаст его. Если такая функция передаст исключение, для соблюдения обещания во время выполнения вызывается функция `terminate()`. Результат прокрутки стека непредсказуем. Таким образом, спецификатор `noexcept` следует использовать в двух случаях: если есть уверенность, что функция не будет передавать исключений, или если совершенно неизвестно, как справиться с ошибкой.

Спецификация запрета передачи исключения фактически обещает *вызывающей стороне* такой функции, что ей не придется иметь дела с исключениями. Функция либо не передаст исключения, либо вся программа закончит работу; в любом случае вызывающей стороне не нести ответственность за исключения.



Во время компиляции компилятор может вообще не проверять спецификации исключения.

Совместимость с прежней версией. Спецификации исключения

У прежних версий языка C++ была более сложная схема спецификаций исключения, позволяющая определять типы исключений, которые могла бы передавать функция. Функция может определить ключевое слово `throw`, сопровождаемое заключенным в скобки списком типов, которые могла бы передать функция. Спецификатор `throw` располагается в том же месте, где и спецификатор `noexcept` в текущем языке.

Этот подход никогда широко не использовался и не рекомендован в текущем стандарте. Хотя один случай использования более сложной старой схемы распространен довольно широко. Функция, обозначенная как `throw()`, обещает не передавать никаких исключений:

```
void recoup( int ) noexcept; // recoup() не передает ничего
void recoup( int ) throw(); // эквивалентное
объявление
```

Эти объявления функции `recoup()` эквивалентны. Оба указывают, что функция `recoup()` не будет передавать исключений.

Аргументы спецификации noexcept

Спефикатор `noexcept` получает необязательный аргумент, тип которого должен быть преобразуем в тип `bool`: если аргументом будет `true`, то функция не будет передавать исключений; если `false` — то может:

```
void recoup( int ) noexcept( true ); // не будет
передавать исключений
void alloc( int ) noexcept( false ); // может
передавать исключения
```

Оператор noexcept



Аргументы спецификатора `noexcept` зачастую создаются с использованием *оператора noexcept*. Оператор `noexcept` — унарный, возвращающий константное логическое выражение `r`-значения, означающее способность данного выражения передавать исключения. Подобно оператору `sizeof` (см. раздел 4.9), оператор `noexcept` не

вычисляет свой операнд.

Например, следующее выражение возвращает значение `true`:

```
noexcept( recoup( i ) ) // true, если вызов функции  
recoup() не может
```

// передать исключение, и false

в противном случае

поскольку функция `recoup()` объявлена со спецификатором `noexcept`. В более общем виде выражение `noexcept(e)` возвращает значение `true`, если у всех вызванных `e` функций нет спецификаций передачи и сама `e` не содержит операторов `throw`. В противном случае выражение `noexcept(e)` возвращает значение `false`.

Оператор `noexcept` можно использовать для формирования спецификатора исключения следующим образом:

```
void f() noexcept( noexcept( g() ) ); // f() имеет тот же спецификатор
```

// исключения,

что и `g()`

Если функция `g()` обещает не передавать исключений, то `f()` также не будет. Если `g()` не имеет спецификатора исключения или имеет спецификатор, позволяющий передачу исключений, то функция `f()` также может передавать их.



Ключевое слово `noexcept` имеет два значения: это спецификатор исключения, когда оно следует за списком параметров функции, и оператор, который зачастую используется как логический аргумент для спецификатора исключения `noexcept`.

Спецификации исключения и указатели, виртуальные функции, функции управления копированием

Хотя спецификатор `noexcept` не является частью типа функции, наличие у функции спецификатора исключения влияет на ее использование.

Указатель на функцию и функция, на которую указывает этот указатель, должны иметь одинаковые спецификации. Таким образом, если

объявлен указатель со спецификатором запрета передачи исключения, то использовать этот указатель можно только для указания на функции с подобным спецификатором. Указатель на функцию, способную передавать исключение, определенный явно или неявно, может указывать на любую функцию, даже если она обещает не передавать исключения:

```
// recoup() и pf1() обещают не передавать исключений
void (*pf1)(int) noexcept = recoup;
// ok: recoup() не будет передавать исключений; и не имеет значения,
// что pf2() может
void (*pf2)(int) = recoup;
pf1 = alloc; // ошибка: alloc() может передать исключение, но pf1()
              // обещала, что не будет
pf2 = alloc; // ok: pf2() и alloc() могли бы передать исключение
```

Если виртуальная функция обещает не передавать исключений, унаследованные виртуальные функции также должны обещать не передавать исключений. С другой стороны, если базовая функция позволяет передачу исключения, то производным функциям стоит быть ограниченным строже и обещать не передавать их:

```
class Base {
public:
    virtual double f1(double) noexcept; // не передает исключения
    virtual int f2() noexcept(false); // может передавать
    virtual void f3(); // может передавать
};

class Derived : public Base {
public:
    double f1(double); // ошибка: Base::f1()
    // обещает не передавать
    int f2() noexcept(false); // ok: та же спецификация, как у Base::f2()
    void f3() noexcept; // ok: Derived::f3()
    // ограничена строже
```

```
};
```

Когда компилятор синтезирует функции-члены управления копированием, он создает для них спецификацию исключения. Если все соответствующие функции-члены всех базовых классов обещают не передавать исключений, то синтезируемые функции-члены также будут `noexcept`. Если какая-нибудь функция, вызванная синтезируемым членом, может передать исключение, то этот синтезируемый член помечается как `noexcept(false)`. Кроме того, если разработчик не предоставил спецификацию исключения для деструктора, который он определяет, компилятор синтезирует ее сам. Компилятор создает ту же спецификацию, которую он создал бы, будь то синтезируемый деструктор для этого класса.

Упражнения раздела 18.1.4

Упражнение 18.8. Пересмотрите написанные классы и добавьте соответствующие спецификации исключения к их конструкторам и деструкторам. Если вы полагаете, что некоторые из ваших деструкторов могли бы передавать исключения, изменить код так, чтобы это было невозможно.

18.1.5. Иерархии классов исключений

Классы исключений (см. раздел 5.6.3) стандартной библиотеки формируют иерархию наследования (см. главу 15), представленную на рис. 18.1.

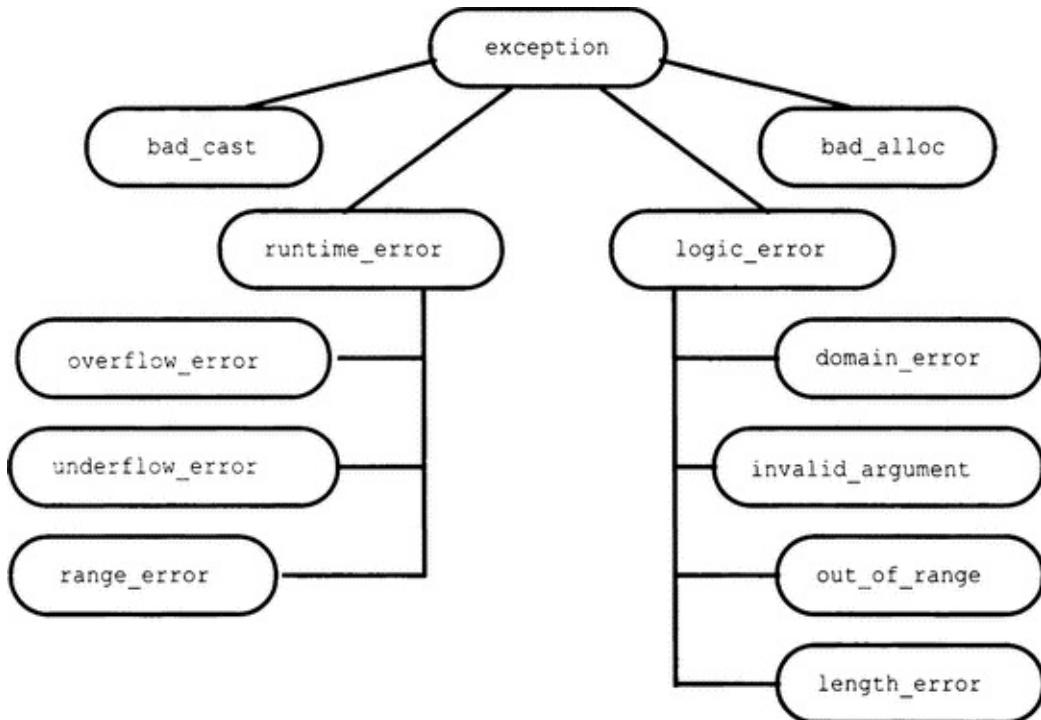


Рис. 18.1. Иерархия классов исключений стандартной библиотеки

Единственными функциями, определенными типом `exception`, являются конструктор копий, оператор присвоения копий, виртуальный деструктор и виртуальная функция-член `what()`. Она возвращает указатель типа `const char*` на символьный массив с нулевым символом в конце и, как гарантируется, не передает никаких исключений.

Классы исключений `exception`, `bad_cast` и `bad_alloc` определяют также стандартный конструктор. Классы `runtime_error` и `logic_error` не имеют стандартного конструктора, но имеют конструкторы, получающие символьную строку в стиле С или аргумент библиотечного типа `string`. Эти аргументы предназначены для дополнительной информации об ошибке. Функция `what()` этих классов возвращает сообщение, использованное для инициализации объекта исключения. Поскольку функция `what()` виртуальная, при обработке ссылки на базовый тип вызов функции `what()` выполнит ту версию, которая соответствует динамическому типу объекта исключения.

Классы исключения для приложения книжного магазина

В приложениях иерархию исключений зачастую дополняют, определяя классы, производные от класса `exception` (или другого библиотечного класса, производного от него). Такие классы представляют исключения,

специфические для данного приложения.

Если бы предстояло создать реальное приложение книжного магазина, его классы были бы гораздо сложнее, чем в примерах этой книги. Одной из причин усложнения является обработка исключений. Фактически пришлось бы создать собственную иерархию исключений, отражающую вероятные проблемы, специфические для данного приложения. В этом проекте могли бы понадобиться следующие классы:

```
// гипотетический класс исключения для приложения
книжного магазина
class out_of_stock: public std::runtime_error {
public:
    explicit out_of_stock(const std::string &s):
        std::runtime_error(s) { }
};

class isbn_mismatch: public std::logic_error {
public:
    explicit isbn_mismatch(const std::string &s):
        std::logic_error(s) { }
    isbn_mismatch(const std::string &s,
                  const std::string &lhs, const std::string &rhs):
        std::logic_error(s), left(lhs), right(rhs) { }
    const std::string left, right;
};
```

Здесь специфические для приложения классы исключения определены как производные от стандартного класса исключения. Любую иерархию классов, включая иерархию исключений, можно рассматривать как слоистую структуру. По мере углубления иерархии каждый слой становится более специализированным. Например, первым и наиболее общим слоем иерархии является класс `exception`. При получении объекта этого типа будет известно только то, что в приложении произошла какая-то ошибка.

Второй слой специализирует исключение на две обширные категории: ошибки времени выполнения и логические ошибки. Ошибки времени выполнения могут быть обнаружены только при запуске программы. Логические ошибки, в принципе, могут быть обнаружены в приложении.

Классы исключений книжного магазина представляют даже более специализированный слой. Класс `out_of_stock` представляет проблему времени выполнения, специфическую для данного приложения. Он используется для оповещения о нарушении порядка выполнения. Класс

исключения `isbn_mismatch` представляет собой более специализированную форму класса `logic_error`. В принципе программа может обнаружить несоответствие ISBN, вызвав функцию `isbn()`.

Использование собственных типов исключений

Собственные классы исключений применяются точно так же, как и классы стандартной библиотеки. Одна часть программы передает объект одного из этих классов, а другая получает и обрабатывает его, устранивая проблему. Например, для перегруженного оператора суммы класса `Sales_item` можно создать класс исключения `isbn_mismatch`, передаваемого в случае обнаружения ошибки несовпадения ISBN.

```
// передает исключение, если isbn объектов не совпадают
```

```
Sales_data&
Sales_data::operator+=( const Sales_data& rhs) {
    if (isbn() != rhs.isbn())
        throw isbn_mismatch("wrong isbns", isbn(),
rhs.isbn());
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}
```

Обнаружив эту ошибку, использующий оператор `+=` код сможет передать соответствующее сообщение об ошибке и продолжить работу.

```
// применение исключения в приложении книжного магазина
```

```
Sales_data item1, item2, sum;
while (cin >> item1 >> item2) { // прочитать две транзакции
    try {
        sum = item1 + item2; // вычислить их сумму
        // использовать сумму
    } catch (const isbn_mismatch &e) {
        cerr << e.what() << ": left isbn( " << e.left
            << " ) right isbn ( " << e.right << " ) " <<
endl;
    }
}
```

Упражнения раздела 18.1.5

Упражнение 18.9. Определите описанные в этом разделе классы исключений приложения книжного магазина и перепишите составной оператор присвоения класса `Sales_data` так, чтобы он передавал исключение.

Упражнение 18.10. Напишите программу, использующую оператор суммы класса `Sales_data` для объектов с разными ISBN. Напишите две версии программы: способную обрабатывать исключения и не обрабатывающую их. Сравните поведение программ, чтобы ознакомиться с тем, что происходит при отсутствии обработки исключения.

Упражнение 18.11. Почему так важно, чтобы функция `what()` не передавала исключений?

18.2. Пространства имен

В больших программах обычно используют библиотеки от независимых разработчиков. В таких библиотеках обычно определено множество глобальных имен классов, функций и шаблонов. Когда приложение использует библиотеки от многих разных поставщиков, некоторые из этих имен почти неизбежно совпадут. Библиотеки, помещающие имена в глобальное пространство имен, вызывают *загромождение пространства имен* (namespace pollution).

Традиционно программисты избегают загромождения пространства имен, используя для глобальных сущностей очень длинные имена, зачастую содержащие префикс, означающий библиотеку, в которой определено имя:

```
class cplusplus_primer_Query { ... };  
string          cplusplus_primer_make_plural(size_t,  
string&);
```

Это решение далеко от идеала: программистам неудобно писать и читать программы, использующие длинные имена.

Пространства имен (namespace) предоставляют намного более контролируемый механизм предотвращения конфликтов имени. Пространства имен разделяют глобальное пространство имен. Пространство имен — это область видимости. При определении имен библиотеки в пространстве имен, авторы (и пользователи) библиотеки могут избежать ограничений, присущих глобальным именам.

18.2.1. Определение пространств имен

Определение пространства имен начинается с ключевого слова `namespace`, сопровождаемого именем пространства имен. После имени пространства имен следуют заключенные в фигурные скобки объявление и определения. В пространство имен может быть помещено любое объявление, которое способно присутствовать в глобальной области видимости, включая классы, переменные (с инициализацией), функции (с их определениями), шаблоны и другие пространства имен.

```
namespace cplusplus_primer {
    class Sales_data { /* ... */;
    Sales_data operator+(const Sales_data&,
                          const Sales_data&);
    class Query { /* ... */;
    class Query_base { /* ... */;
} // подобно блокам, пространства имен не
завершаются точкой с запятой
```

Этот код определяет пространство имен `cplusplus_primer` с четырьмя членами: тремя классами и перегруженным оператором `+`.

Подобно другим именам, имя пространства имен должно быть уникальным в той области видимости, в которой оно определено. Пространства имен могут быть определены в глобальной области видимости или в другом пространстве имен. Они не могут быть определены в функциях или классах.



Область видимости пространства имен не заканчивается точкой с запятой.

Каждое пространство имен является областью видимости

Как и в случае любой области видимости, каждое имя в пространстве имен должно относиться к уникальной сущности в пределах данного пространства имен. Поскольку разные пространства имен вводят разные области видимости, в разных пространствах имен могут быть члены с одинаковым именем.

К именам, определенным в пространстве имен, другие члены данного пространства имен могут обращаться непосредственно, включая области

видимости, вложенные в пределах этих членов. Код вне пространства имен должен указывать пространство имен, в котором определено имя:

```
cplusplus_primer::Query q =  
    cplusplus_primer::Query("hello");
```

Если другое пространство имен (например, AddisonWesley) тоже содержит класс `Query` и этот класс необходимо использовать вместо определенного в пространстве имен `cplusplus_primer`, приведенный выше код придется изменить следующим образом:

```
AddisonWesley::Query q =  
AddisonWesley::Query("hello");
```

Пространства имен могут быть разобщены

Как упоминалось в разделе 16.5, в отличие от других областей видимости, пространство имен может быть определено в нескольких частях. Вот определение пространства имен:

```
namespace nsp {  
    // объявления  
}
```

Этот код определяет новое пространство имен `nsp` или добавляет члены к уже существующему. Если пространство имен `nsp` еще не определено, то создается новое пространство имен с этим именем. В противном случае это определение открывает уже существующее пространство имен и добавляет в него новые объявления.

Тот факт, что определения пространств имен могут быть разобщены, позволяет составить пространство имен из отдельных файлов интерфейса и реализации. Таким образом, пространство имен может быть организовано таким же образом, как и определения собственных классов или функций.

- Члены пространства имен, являющиеся определениями классов, объявлениями функций и объектов, составляющих часть интерфейса класса, могут быть помещены в файлы заголовка. Эти заголовки могут быть подключены в те файлы, которые используют эти члены пространства имен.

- Определения членов пространства имен могут быть помещены в отдельные файлы исходного кода.

Организовав пространство имен таким образом, можно также удовлетворить требование, согласно которому различные сущности, включая не подлежащие встраиванию функции, статические переменные-члены, переменные и т.д., должны быть определены в программе только один раз. Это требование распространяется и на имена, определенные в

пространстве имен. Отделив интерфейс и реализацию, можно гарантировать, что имена функций и другие имена будут определены только один раз и именно это объявление будет многократно использоваться впоследствии.



Рекомендуем

Для представления несвязанных типов в составных пространствах имен следует использовать отдельные файлы.

Определение пространства имен с `cplusplus_primer`

Используя эту стратегию для отделения интерфейса от реализации, определим библиотеку `cplusplus_primer` в нескольких отдельных файлах. Объявления класса `Sales_data` и связанных с ним функций поместим в файл заголовка `Sales_data.h`, а таковые для класса `Query` (см. главу 15) — в заголовок `Query.h` и т.д. Соответствующие файлы реализации были бы в таких файлах, как `Sales_data.cc` и `Query.cc`:

```
// ---- Sales_data.h ----
// директивы #include должны быть перед открытием
// пространства имен
#include <string>
namespace cplusplus_primer {
    class Sales_data { /* ... */;
        Sales_data operator+(const Sales_data&,
                              const Sales_data&);
    // объявления остальных функций интерфейса класса
    Sales_data
    }
// ---- Sales_data.cc ----
// все директивы #include перед открытием
// пространства имен
#include "Sales_data.h"
namespace cplusplus_primer {
    // определения членов класса Sales_data и
    // перегруженных операторов
}
```

Использующая эту библиотеку программа включила бы все необходимые заголовки. Имена в этих заголовках определены в

пространстве имен `cplusplus_primer`:

```
// ---- user.cc ----
// имена заголовка Sales_data.h находятся в
пространстве
// имен cplusplus_primer
#include "Sales_data.h"
int main() {
    using cplusplus_primer::Sales_data;
    Sales_data trans1, trans2;
    // ...
    return 0;
}
```

Подобная организация программы придает библиотеке свойство модульности, необходимое как разработчикам, так и пользователям. Каждый класс организован в виде двух файлов: интерфейса и реализации. Пользователь одного класса вовсе не должен использовать при компиляции другие классы. Их реализацию можно скрыть от пользователей, разрешив при этом компилировать и компоновать файлы `Sales_data.cc` и `user.cc` в одну программу, причем без опасений по поводу возникновения ошибок во время компиляции или компоновки. Кроме того, разработчики библиотеки могут работать над реализацией каждого класса независимо.

В использующую эту библиотеку программу следует подключить все необходимые заголовки. Имена в этих заголовках определены в пространстве имен `cplusplus_primer`.

Следует заметить, что директивы `#include` обычно не помещают в пространство имен. Если попробовать сделать это, то произойдет попытка определения всех имен в этом заголовке как членов окружающего пространства имен. Например, если бы файл `Sales_data.h` открыл пространство имен `cplusplus_primer` прежде, чем включить заголовок `string`, то в программе была бы ошибка, поскольку это привело бы к попытке определить пространство имен `std` в пространстве имен `cplusplus_primer`.

Определение членов пространства имен

Если объявления находятся в области видимости, то код в пространстве имен может использовать короткую форму имен, определенных в том же (или вложенном) пространстве имен:

```
#include "Sales_data.h"
namespace cplusplus_primer { // повторное открытие
    // члены, определенные в пространстве имен, могут
    // использовать имена
    // без уточнений
    std::istream&
    operator>>( std::istream& in, Sales_data& s) { /* ...
... */ }
}
```

Член пространства имен может быть также определен вне определения пространства имен. Для этого применяется подход, подобный определению членов класса вне его. Объявление пространства имен должно находиться в области видимости, а в определении следует указать пространство имен, которому принадлежит имя.

```
// члены пространства имен, определенные вне его,
// должны использовать
// полностью квалифицированные имена
cplusplus_primer::Sales_data
cplusplus_primer::operator+( const Sales_data& lhs,
                           const Sales_data& rhs)
{
    Sales_data ret(lhs);
    // ...
}
```

Подобно членам класса, определенным вне самого класса, когда встречается полностью определенное имя, оно находится в пределах пространства имен. В пространстве имен `cplusplus_primer` можно использовать другие имена членов пространства имен без квалификации. Таким образом, хотя класс `Sales_data` является членом пространства имен `cplusplus_primer`, для определения параметров его функций можно использовать его имя без квалификации.

Хотя член класса пространства имен может быть определен вне его определения, такие определения должны присутствовать в окружающем пространстве имен. Таким образом, оператор `operator+` класса `Sales_data` можно определить в пространстве имен `cplusplus_primer` или в глобальной области видимости. Но он не может быть определен в несвязанном пространстве имен.

Специализация шаблона

Специализация шаблона должна быть определена в том же пространстве имен, которое содержит первоначальный шаблон (см. раздел 16.5). Подобно любым другим именам пространства имен, пока специализация объявлена в пространстве имен, ее можно определить вне пространства имен:

```
// специализацию нужно объявить как член
пространства std
namespace std {
    template <> struct hash<Sales_data>;
}
// добавив объявление для специализации к
пространству std,
// специализацию можно определить вне пространства
имен std
template <> struct std::hash<Sales_data> {
    size_t operator()(const Sales_data& s) const {
        return hash<string>()(s.bookNo) ^
               hash<unsigned>()(s.units_sold) ^
               hash<double>()(s.revenue);
    }
    // другие члены как прежде
};
```

Глобальное пространство имен

Имена, определенные в глобальной области видимости (т.е. имена, объявленные вне любого класса, функции или пространства имен), определяются в *глобальном пространстве имен* (global namespace). Глобальное пространство имен неявно объявляется и существует в каждом приложении. Каждый файл, который определяет сущность в глобальной области видимости (неявно), добавляет ее имя к глобальному пространству имен.

Для обращения к членам глобального пространства имен применяется *оператор области видимости* (оператор `::`) (scope operator). Поскольку глобальное пространство имен неявно, у него нет имени.

Форма записи при обращении к члену глобального пространства имен имеет следующий вид.

`:: член_имя`

Вложенные пространства имен

Вложенное пространство имен (nested namespace) — это пространство имен, определенное в другом пространстве имен:

```
namespace cplusplus_primer {
    // первое вложенное пространство имен: определение
    // части
    // библиотеки Query
    namespace QueryLib {
        class Query { /* ... */ };
        Query operator&(const Query&, const Query&);
        // ...
    }
    // второе вложенное пространство имен: определение
    // части
    // библиотеки Sales_data
    namespace Bookstore {
        class Quote { /* ... */ };
        class Disc_quote : public Quote { /* ... */ };
        // ...
    }
}
```

Вложенное пространство имен — это вложенная область видимости, ее область видимости вкладывается в пределы содержащего ее пространства имен. Имена вложенных пространств имен подчиняются обычным правилам: имена, объявленные во внутреннем пространстве имен, скрывают объявления того же имени во внешнем пространстве. Имена, определенные во вложенном пространстве имен, являются локальными для внутреннего пространства имен. Код во внешних частях окружающего пространства имен может обратиться к имени во вложенном пространстве имен только через его квалифицированное имя. Например, имя класса `QueryLib`, объявленного во вложенном пространстве имен, выглядит следующим образом:

```
cplusplus_primer::QueryLib::Query
```

Встраиваемые пространства имен



Новый стандарт ввел новый вид вложенного пространства имен — *встраиваемое пространство имен* (inline namespace). В отличие от

обычных вложенных пространств имен, имена из встраиваемого пространства имен применяются так, как будто они являются непосредственными членами окружающего пространства имен. Таким образом, нет необходимости в квалификации имен из встраиваемого пространства имен. Для доступа к ним достаточно использовать имя окружающего пространства имен.

Для определения встраиваемого пространства имен ключевое слово `namespace` предваряется ключевым словом `inline`:

```
inline namespace FifthEd {  
    // пространство имен для кода Primer Fifth Edition  
}  
namespace FifthEd { // неявно встраиваемая  
    class Query_base { /* ... */;  
        // другие объявления, связанные с классом Query  
    }
```

Это ключевое слово должно присутствовать в первом определении пространства имен. Если пространство имен вновь открывается позже, ключевое слово `inline` не обязательно, но может быть повторено.

Встраиваемые пространства имен зачастую используются при изменении кода от одного выпуска приложения к следующему. Например, весь код текущего издания *Вводного курса* можно поместить во встраиваемое пространство имен. Код предыдущих версий был бы в обычных, а не встраиваемых пространствах имен:

```
namespace FourthEd {  
    class Item_base { /* ... */;  
    class Query_base { /* ... */;  
        // другой код из Fourth Edition  
    }
```

Общее пространство имен `cplusplus_primer` включило бы определения обоих пространств имен. Например, с учетом того, что каждое пространство имен было определено в заголовке с соответствующим именем, пространство имен `cplusplus_primer` можно определить следующим образом:

```
namespace cplusplus_primer {  
    #include "FifthEd.h"  
    #include "FourthEd.h"  
}
```

Поскольку пространство имен `FifthEd` встраиваемое, код

обращающийся к имени из пространства имен `cplusplus_primer::`, получит версию из этого пространства имен. Если понадобится код прежнего издания, к нему можно обратиться как к любому другому вложенному пространству имен, указав все имена окружающих пространств имен, например:

```
cplusplus_primer::FourthEd::Query_base.
```

Безымянные пространства имен

У *безымянного пространства имен* (*unnamed namespace*) сразу за ключевым словом `namespace` следует блок объявлений, разграниченных фигурными скобками. У переменных, определенных в безымянном пространстве имен, статическая продолжительность существования: они создаются перед их первым использованием и удаляются по завершении программы.

Безымянное пространство имен может быть разобщено в пределах данного файла, но не охватывающих файлов. У каждого файла есть собственное безымянное пространство имен. Если два файла содержат безымянные пространства имен, эти пространства имен не связаны. Оба безымянных пространства имен могут определить одинаковое имя, и эти определения будут относиться к разным сущностям. Если заголовок определяет безымянное пространство имен, то имена в этом пространстве определяют сущности, локальные для каждого файла, включенного в заголовок.



В отличие от других пространств имен, безымянное пространство является локальным для конкретного файла и никогда не охватывает несколько файлов.

Имена, определенные в безымянном пространстве имен, используются непосредственно; в конце концов, для их квалификации нет никакого имени пространства имен. Для обращения к членам безымянных пространств имен невозможно использовать оператор области видимости.

Имена, определенные в безымянном пространстве имен, находятся в той же области видимости, что и область видимости, в которой определено пространство имен. Если безымянное пространство имен определяется в наиболее удаленной области видимости файла, то имена в безымянном

пространстве имен должны отличаться от имен, определенных в глобальной области видимости:

```
int i; // глобальное объявление для i
namespace {
    int i;
}
// неоднозначность: определено глобально и в не
вложенном, безымянном
// пространстве имен
i = 10;
```

Во всем остальном члены безымянного пространства имен являются обычными сущностями программы. Безымянное пространство имен, как и любое другое пространство имен, может быть вложено в другое пространство имен. Если безымянное пространство имен вкладывается, то к содержащимся в нем именам обращаются обычным способом, используя имена окружающего пространства имен:

```
namespace local {
    namespace {
        int i;
    }
}
// ok: i определено во вложенном безымянном
пространстве имен
// отдельно от глобального i
local::i = 42;
```

Безымянные пространства имен вместо статических файловых объектов

До введения пространств имен в стандарт C++, чтобы сделать имена локальными для файла, их приходилось объявлять статическими (*static*). Применение *статических файловых объектов* (*file static*) унаследовано от языка С. В языке С объявленный статическим глобальный объект был невидим вне того файла, в котором он объявлен.



ВНИМАНИЕ

В соответствии со стандартом C++ применение объявлений статических файловых объектов не рекомендуется. Вместо них используются

безымянные пространства имен.

Упражнения раздела 18.2.1

Упражнение 18.12. Организуйте программы, написанные в упражнениях каждой из глав, в их собственные пространства имен. Таким образом, пространство имен `chapter15` содержало бы код для программы запросов, а `chapter10` — код приложения `TextQuery`. Используя эту структуру, откомпилируйте примеры кода приложения `Query`.

Упражнение 18.13. Когда используются безымянные пространства имен?

Упражнение 18.14. Предположим, имеется следующее объявление оператора `operator*`, являющегося членом вложенного пространства имен `mathLib::MatrixLib`:

```
namespace mathLib {  
    namespace MatrixLib {  
        class matrix { /* ... */ };  
        matrix operator*  
            (const matrix &, const matrix &);  
        // ...  
    }  
}
```

Как определить этот оператор в глобальной области видимости?

18.2.2. Использование членов пространства имен

Обращение к члену пространства имен в формате `имя_пространства_имен::имя_члена` является чрезвычайно громоздким, особенно когда имя пространства имен слишком длинное. К счастью, существуют способы, которые облегчают использование имен членов пространства имен. Один из этих способов, объявление `using` (см. раздел 3.1), уже использовался в программах, приведенных выше. Другие способы, псевдонимы пространств имен и директивы `using` будут описаны в этом разделе.

Псевдонимы пространства имен

Псевдоним пространства имен (`namespace alias`) применяется в качестве короткого синонима имени пространства имен. Например, длинное имя пространства имен может иметь следующий вид:

```
namespace cplusplus_primer { /* ... */ };
```

Ему может быть назначен более короткий синоним следующим образом:

```
namespace primer = cplusplus_primer;
```

Объявление псевдонима пространства имен начинается с ключевого слова `namespace`, за которым следует имя псевдонима пространства имен (короткое), сопровожданное знаком `=`, первоначальное имя пространства имен и точка с запятой. Если имя первоначального пространства имен еще не было определено как пространство имен, произойдет ошибка.

Псевдоним пространства имен может быть также применен к вложенному пространству имен:

```
namespace Qlib = cplusplus_primer::QueryLib;  
Qlib::Query q;
```



Пространство имен может иметь множество синонимов или псевдонимов. Все псевдонимы и первоначальное имя пространства имен равнозначны в применении.

Объявления `using` (напоминание)

Имена, представленные в объявлении `using`, подчиняются обычным

правилам области видимости. Имя видимо от точки объявления `using` и до конца области видимости, в которой оно объявлено. Сущности внутренней области видимости скрывают одноименные сущности внешней. Короткие имена могут использоваться только в той области видимости, в которой они объявлены, а также в областях видимости, вложенных в нее. По завершении области видимости следует использовать полные имена.

Объявление `using` может присутствовать в глобальной и локальной области видимости, а также в области видимости пространства имен или класса. Объявление `using` в области видимости класса ограничено именами, определенными в базовом классе определяемого класса (см. раздел 15.5).

Директива `using`

Подобно объявлению `using`, директива `using` (`using directive`) позволяет использовать не квалифицированную форму имен. Однако, в отличие от объявления `using`, здесь не сохраняется контроль над видимостью имен, поскольку все они видимы.

Директива `using` начинается с ключевого слова `using`, за которым следует ключевое слово `namespace`, сопровождаемое именем пространства имен. Если имя пространства не было определено ранее, произойдет ошибка. Директива `using` может присутствовать в глобальной, локальной области видимости или в пространстве имен. Она не может присутствовать в области видимости класса.



ВНИМАНИЕ

Предоставление директив `using` для таких пространств имен, как `std`, которые приложение не контролирует, возвращает все проблемы конфликта имени, присущие использованию нескольких библиотек.

Директива `using` и область видимости

Область видимости имен, указанных директивой `using`, гораздо сложнее, чем в случае объявления `using`. Объявление `using` помещает имя непосредственно в ту же область видимости, в которой находится само объявление `using`. Объявление `using` подобно локальному псевдониму для члена пространства имен.

Директива `using` не объявляет локальные псевдонимы для имен членов пространства имен. Вместо этого она поднимает члены пространства имен в ближайшую область видимости, которая содержит и пространство имен, и саму директиву `using`.

Различие в области видимости между объявлением `using` и директивой `using` проистекает непосредственно из принципа действия этих средств. В случае объявления `using` само имя просто становится доступным в локальной области видимости. Директива `using`, напротив, делает доступным все содержимое пространства имен. Вообще, пространство имен способно включать определения, которые не могут присутствовать в локальной области видимости. Как следствие, директива `using` рассматривается как присутствующая в ближайшей области видимости окружающего пространства имен.

Рассмотрим самый простой случай. Предположим, что в глобальной области видимости определено пространство имен `A` и функция `f()`. Если функция `f()` имеет директиву `using` для пространства имен `A`, функция `f()` будет вести себя так, как будто имена пространства имен `A` присутствовали в глобальной области видимости до определения функции `f()`.

```
// пространство имен A и функция f() определены в
// глобальной области видимости
namespace A {
    int i, j;
}
void f() {
    using namespace A; // переводит имена из области
    // видимости A в
    // глобальную область видимости
    cout << i * j << endl; // использует i и j из
    // пространства имен A
    // ...
}
```

Пример директив `using`

Рассмотрим следующий пример:

```
namespace blip {
    int i = 16, j = 15, k = 23; // другие объявления
```

```

}

int j = 0; // ok: j в пространстве имен blip скрыта
void manip() {
    // директива using; имена пространства имен blip
"добавляются" к
    // глобальной области видимости
    using namespace blip; // конфликт между ::j и
blip::j
    // обнаруживается только при использовании j
    ++i;           // присваивает blip::i значение 17
    ++j;           // ошибка неоднозначности: global j
или blip::j?
    +++;: j;        // ok: присваивает глобальной j
значение 1
    ++blip::j;    // ok: присваивает blip::j значение 16
    int k = 97;   // локальная k скрывает blip::k
    ++k;          // присваивает локальной k значение 98
}

```

Директива `using` в функции `manip()` делает все имена пространства имен `blip` доступными непосредственно. То есть функция `manip()` может обращаться к этим членам, используя краткую форму имен.

Члены пространства имен `blip` выглядят так, как будто они были определены в одной области видимости. Если пространство имен `blip` определено в глобальной области видимости, его члены будут выглядеть так, как будто они объявлены в глобальной области видимости.

Когда пространство имен вводится в окружающую область видимости, имена в пространстве имен вполне могут вступить в конфликт с другими именами, определенными (включенными) в той же области видимости. Например, в функции `manip()` член `j` пространства имен `blip` вступает в конфликт с глобальным объектом `j`. Такие конфликты разрешимы, но для использования имени следует явно указать, какая версия имеется в виду. Любое использование имени `j` в пределах функции `manip()` ведет к неоднозначности.

Чтобы использовать такое имя, как `j`, следует применить оператор области видимости, позволяющий указать требуемое имя. Для указания переменной `j`, определенной в глобальной области видимости, нужно написать `::j`, а для определенной в пространстве имен `blip` — `blip::j`.

Поскольку имена находятся в разных областях видимости, локальные

объявления в пределах функции `manip()` могут скрыть некоторые из имен пространства имен. Локальная переменная `k` скрывает член пространства имен `blip::k`. Обращение к переменной `k` в пределах функции `manip()` вполне однозначно, это обращение к локальной переменной `k`.

Заголовки и объявления `using` или директивы

Заголовок, содержащий директиву или объявление `using` в своей области видимости верхнего уровня, вводит свои имена в каждый файл, который подключает заголовок. Обычно заголовки должны определять только те имена, которые являются частью его интерфейса, но не имена, используемые в его реализации. В результате файлы заголовка не должны содержать директивы или объявлений `using`, кроме как в функциях или пространствах имен (см. раздел 3.1).

Внимание! Избегайте директив `using`

Директивы `using`, вводящие в область видимости все имена из пространства имен, обманчиво просты в использовании. Единственный оператор делает видимыми имена всех членов пространства имен. Хоть этот подход может показаться простым, он создает немало проблем. Если в приложении использовано много библиотек и директива `using` сделает видимыми имена, определенные в них, то вновь возникнет проблема загромождения глобального пространства имен.

Кроме того, не исключено, что при выходе новой версии библиотеки вполне работоспособная в прошлом программа перестанет компилироваться. Причиной этой проблемы может быть конфликт имен новой версии с именами, которые использовались прежде.

Еще одна вызванная директивой `using` проблема неоднозначности обнаруживается только в момент применения. Столь позднее обнаружение означает, что конфликты могут возникать значительно позже применения определенной библиотеки. То есть при использовании в программе новой библиотеки могут возникнуть не обнаруженные ранее конфликты.

Поэтому лучше не полагаться на директиву `using` и использовать объявление `using` для каждого конкретного имени пространства имен, используемого в программе. Это уменьшит количество имен, вводимых в пространство имен. Кроме того, ошибки неоднозначности, причиной которых является объявление `using`, обнаруживаются в точке

объявления, а это существенно упрощает их поиск.



Директивы `using` на самом деле полезны в файлах реализации самого пространства имен.

Упражнения раздела 18.2.2

Упражнение 18.15. Объясните различия между объявлением и директивой `using`.

Упражнение 18.16. Объясните следующий код с учетом того, что объявления `using` для всех членов пространства имен `Exercise` находятся в области, помеченной как *позиция 1*. Что, если вместо этого они располагаются в *позиции 2*? Теперь ответьте на тот же вопрос, но замените объявления `using` директивой `using` для пространства имен `Exercise`.

```
namespace Exercise {  
    int ivar = 0;  
    double dvar = 0;  
    const int limit = 1000;  
}  
int ivar = 0;  
// позиция 1  
void manip() {  
    // позиция 2  
    double dvar = 3.1416;  
    int iobj = limit + 1;  
    ++ivar;  
    ++::ivar;  
}
```

Упражнение 18.17. Напишите код для проверки ответов на предыдущий вопрос.

18.2.3. Классы, пространства имен и области видимости

Поиск имен, используемых в пространстве имен, происходит согласно обычным правилам поиска в языке C++: сначала во внутренней, а затем во внешней области видимости. Имя, используемое в пространстве имен,

может быть определено в одном из окружающих пространств имен, включая глобальное пространство имен. Однако учитываются только те имена, которые были объявлены перед точкой использования в блоках, которые все еще открыты.

```
namespace A {
    int i;
    namespace B {
        int i; // скрывает A::i в B
        int j;
        int f1() {
            int j; // j локальна для f1() и скрывает
A::B::j
            return i; // возвращает B::i
        }
    } // пространство имен B закрыто, и его имена
больше не видимы
    int f2() {
        return j; // ошибка: j не определена
    }
    int j = i; // инициализируется значением A::i
}
```

Когда класс расположен в пространстве имен, процесс поиска остается обычным: когда имя используется функцией-членом, его поиск начинается в самой функции, затем в пределах класса (включающий базовые классы), а потом в окружающих областях видимости, одной или несколькими из которых могли бы быть пространства имен:

```
namespace A {
    int i;
    int k;
    class C1 {
        public:
            C1(): i(0), j(0) { } // ok: инициализирует
C1::i и C1::j
            int f1() { return k; } // возвращает A::k
            int f2() { return h; } // ошибка: h не определена
            int f3();
        private:
            int i; // скрывает A::i в C1
            int j;
```

```

};

int h = i; // инициализируется значением A::i
}

// член f3() определен вне класса C1 и вне
пространства имен A
int A::C1::f3() { return h; } // ok: возвращает
A::h

```

За исключением определений функций-членов, расположенных в теле класса (см. раздел 7.4.1), области видимости всегда просматриваются снизу вверх: имя должно быть объявлено прежде его применения. Следовательно, оператор `return` функции `f2()` не будет откомпилирован. Он попытается обратиться к имени `h` из пространства имен `A`, но там оно еще не определено. Если бы это имя `h` было определено в пространстве имен `A` прежде определения класса `C1`, его использование было бы вполне допустимо. Аналогично использование имени `h` в функции `f3()` вполне допустимо, поскольку функция `f3()` определена уже после определения `A::h`.



Порядок просмотра областей видимости при поиске имени определяется по полностью квалифицированному имени функции. Полностью квалифицированное имя указывает в обратном порядке области видимости, в которых происходит поиск.

Спецификаторы `A::C1::f3()` указывают обратный порядок, в котором просматриваются области видимости класса и пространств имен. Первая область видимости — это функция `f3()`. Далее следует область видимости ее класса `C1`. Область видимости пространства имен `A` просматривается в последнюю очередь, перед переходом к области видимости, содержащей определение функции `f3()`.



Зависимый от аргумента поиск и параметры типа класса

Рассмотрим простую программу:

```
std::string s;
```

```
std::cin >> s;
```

Как известно, этот вызов эквивалентен следующему (см. раздел 14.1):

```
operator>>( std::cin, s );
```

Функция `operator>>` определена библиотекой `string`, которая в свою очередь определяется в пространстве имен `std`. Но все же оператор `>>` можно вызывать без спецификатора `std::` и без объявления `using`.

Непосредственно обратиться к оператору вывода можно потому, что есть важное исключение из правила скрытия имен, определенных в пространстве имен. Когда объект класса передается функции, компилятор ищет пространство имен, в котором определяется класс аргумента *в дополнение* к обычному поиску области видимости. Это исключение применимо также к вызовам с передачей указателей или ссылок на тип класса.

В этом примере, когда компилятор встречает "вызов" оператора `operator>>`, он ищет соответствующую функцию в текущей области видимости, включая области видимости, окружающие оператор вывода. Кроме того, поскольку выражение вывода имеет параметры типа класса, компилятор ищет также в пространствах имен, в которых определяются типы `cin` и `s`. Таким образом, для этого вызова компилятор просмотрит пространство имен `std`, определяющее типы `istream` и `string`. При поиске в пространстве имен `std` компилятор находит функцию вывода класса `string`.

Это исключение из правил поиска позволяет функции, не являющейся членом класса, быть концептуально частью интерфейса к классу и использоваться без отдельного объявления `using`. Без этого исключения из правил поиска для оператора вывода всегда пришлось бы предоставлять соответствующее объявление `using`:

```
using std::operator>>; // чтобы позволить cin >> s
```

Либо пришлось бы использовать форму записи вызова функции, включающую спецификатор пространства имен:

```
std::operator>>( std::cin, s ); // ok: явное
```

использование `std::>>`

Не было бы никакого способа использовать синтаксис оператора. Любое из этих объявлений выглядит неуклюже и существенно затруднило бы использование библиотеки ввода-вывода.

Поиск и функции `std::move()` и `std::forward()`

Многим, возможно, даже большинству программистов C++ никогда не понадобится зависимый от аргумента поиск. Обычно, если приложение определяет имя, уже определенное в библиотеке, истинно одно из двух: либо обычные правила перегрузки определят, относится ли данный конкретный вызов к библиотечной версии функции, или к версии приложения, или приложение никогда не сможет использовать библиотечную функцию.

Теперь рассмотрите библиотечные функции `move()` и `forward()`. Обе являются шаблонами функций, и библиотека определяет их версии с одним параметром функции в виде ссылки на `r`-значение. Как уже упоминалось, параметру ссылки на `r`-значение в шаблоне функции может соответствовать любой тип (см. раздел 16.2.6). Если приложение определяет функцию по имени `move()`, получающую один параметр, то (вне зависимости от типа параметра) версия функции `move()` из приложения вступит в конфликт с библиотечной версией. Это справедливо и для функции `forward()`.

В результате конфликты имен для функций `move()` (и `forward()`) более вероятны, чем для других библиотечных функций. Кроме того, поскольку функции `move()` и `forward()` осуществляют весьма специфические для типа манипуляции, вероятность того, что в приложении специально необходимо переопределить поведение этих функций, довольно мала.

Тот факт, что конфликты имен с этими функциями более вероятны (и менее вероятно, что намеренными), объясняет, почему их имена всегда следует использовать полностью квалифицированными (см. раздел 12.1.5). Форма записи `std::move()`, а не просто `move()` гарантирует применение версии из стандартной библиотеки.



Дружественные объявления и зависимый от аргумента поиск

Напомним, что на момент, когда класс объявляет функцию дружественной (см. раздел 7.2.1), объявление функции необязательно должно быть видимым. Если объявление функции еще не видимо, результатом объявления ее дружественной окажется помещение объявления данной функции или класса в окружающую область видимости. Комбинация этого правила и зависимого от аргумента поиска может привести к неожиданным результатам:

```

namespace A {
    class C {
        // два друга; ничего не объявлено кроме
дружественных отношений
        // эти функции неявно являются членами
пространства имен A
        friend void f2(); // не будет найдено, если не
объявлено иное
        friend void f(const C&); // найдено зависимым от
аргумента
                                // поиском
    };
}

```

Здесь функции `f()` и `f2()` являются членами пространства имен `A`. Зависимый от аргумента поиск позволяет вызвать функцию `f()`, даже если для нее нет никакого дополнительного объявления:

```

int main() {
    A::C cobj;
    f(cobj); // ok: находит A::f() по объявлению
дружественным в A::C
    f2(); // ошибка: A::f2() не объявлена
}

```

Поскольку функция `f()` получает аргумент типа класса и неявно объявляется в том же пространстве имен, что и `C`, при вызове она будет найдена. Так как у функции `f2()` никакого параметра нет, она не будет найдена.

Упражнения раздела 18.2.3

Упражнение 18.18. С учетом следующего типичного определения функции `swap()` в разделе 13.3 определите, какая ее версия используется, если `mem1` имеет тип `string`. Что, если `mem1` имеет тип `int?` Объясните, как будет проходить поиск имен в обоих случаях.

```

void swap( T v1, T v2) {
    using std::swap;
    swap( v1.mem1, v2.mem1);
    // обмен остальных членов типа T
}

```

Упражнение 18.19. Что, если бы вызов функции `swap()` был бы таким

```
std::swap( v1. mem1, v2. mem1 ) ?
```

18.2.4. Перегрузка и пространства имен

Пространства имен могут повлиять на подбор функции (см. раздел 6.4) двумя способами. Один из них вполне очевиден: объявление или директива `using` может добавить функцию в набор кандидатов. Второй способ менее очевиден.



Зависимый от аргумента поиск и перегрузка

Как упоминалось в предыдущем разделе, поиск имен функций, имеющих один или несколько аргументов типа класса, осуществляется также и в пространстве имен, в котором определен класс каждого аргумента. Это правило влияет также и на выбор кандидатов. Каждое пространство имен, в котором определен класс, используемый в качестве типа параметра (а также те, в которых определены его базовые классы), участвует в поиске функции-кандидата. Все функции этих пространств имен, которые имеют имя, совпадающее с использованным при вызове, будут добавлены в набор кандидатов. Эти функции будут добавлены даже тогда, когда *они не видимы в точке обращения*:

```
namespace NS {  
    class Quote { /* ... */ };  
    void display(const Quote&) { /* ... */ }  
}  
// Базовый класс Bulk_item объявлен в пространстве  
имен NS  
class Bulk_item : public NS::Quote { /* ... */ };  
int main()  
{  
    Bulk_item book1;  
    display(book1);  
    return 0;  
}
```

Аргумент `book1` функции `display()` имеет тип класса `Bulk_item`. Функциями-кандидатами для этого вызова функции `display()` будут не только функции с объявлениями, видимыми на момент вызова, но и те, которые объявлены в пространстве имен класса `Bulk_item` и его базового

класса `Quote`. Таким образом, функция `display(const Quote&)`, объявленная в пространстве имен `NS`, будет добавлена в набор функций кандидатов.

Перегрузка и объявления using

Чтобы уяснить взаимодействие объявлений `using` и перегрузки, важно помнить, что объявление `using` объявляет только имя, а не конкретную функцию (см. раздел 15.6):

```
using NS::print( int ); // ошибка: нельзя указать
список параметров
using NS::print;           // ok: в объявлении using
указывают только имена
```

Когда объявление `using` используется для функции, все версии этой функции переводятся в текущую область видимости.

Объявление `using` подключает все версии перегруженной функции, чтобы не нарушить интерфейс пространства имен. Ведь предоставляя разные версии функции, автор библиотеки имел на то весомую причину. Разрешив пользователям игнорировать некоторые (но не все) функции из набора перегруженных версий, можно получить довольно странное поведение программы.

Функции, предоставленные объявлением `using`, перегружают любые другие объявления одноименных функций, уже находящихся в данной области видимости.

Если объявление `using` расположено в локальной области видимости, эти имена скрывают существующие объявления для того имени во внешней области видимости. Если объявление `using` вводит функцию в область видимости, в которой уже есть функция с тем же именем и тем же списком параметров, объявление `using` окажется ошибочным. В противном случае объявление `using` создаст дополнительный перегруженный экземпляр данной функции. В результате набор функций-кандидатов увеличится.

Перегрузка и директивы using

Директива `using` переводит члены пространства имен в окружающую область видимости. Если имя функции пространства имен совпадает с именем функции той области видимости, в которую помещено пространство имен, эта функция будет добавлена в набор перегруженных функций.

```

namespace libs_R_us {
    extern void print(int);
    extern void print(double);
}
// обычное объявление
void print(const std::string &);

// директива using добавила имена в набор функций-
кандидатов для вызова
// функции print():
using namespace libs_R_us;
// кандидатами на вызов print() в настоящий момент
являются:
// print(int) from libs_R_us
// print(double) from libs_R_us
// print(const std::string &) declared explicitly
void fooBar(int ival) {
    print("Value: "); // вызов глобальной print(const string &)
    print(ival); // вызов libs_R_us::print(int)
}

```

В отличие от объявления *using*, не будет ошибки, если директива *using* предоставит функцию с теми же параметрами, что и у существующей функции. Подобно другим конфликтам, вызванным директивами *using*, не будет никаких проблем, если не пытаться вызывать функцию без уточнения, относится ли она к пространству имен или к текущей области видимости.

*Перегрузка при нескольких директивах *using**

Если в коде присутствует несколько директив *using*, частью набора функций-кандидатов станут соответствующие функции из каждого пространства имен.

```

namespace AW {
    int print(int);
}

namespace Primer {
    double print(double);
}

// директивы using создают набор перегруженных
функций из разных

```

```

// пространство имен
using namespace AW;
using namespace Primer;
long double print( long double );
int main() {
    print(1); // вызов AW::print( int )
    print( 3.1 ); // вызов Primer::print( double )
    return 0;
}

```

Набор перегруженных функций `print()` в глобальной области видимости содержит функции `print(int)`, `print(double)` и `print(long double)`. Все они составят набор перегруженных функций, рассматриваемых при вызове функции `print()` в функции `main()`, даже в том случае, если первоначально эти функции были объявлены в различных областях видимости пространства имен.

Упражнения раздела 18.2.4

Упражнение 18.20. С учетом следующего кода укажите, какие из функций (если они есть) соответствуют обращению к функции `compute()`. Перечислите функции-кандидаты и подходящие функции. Какая последовательность преобразований типов (если есть) будет применена к аргументу, чтобы он точно соответствовал параметру каждой подходящей функции?

```

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 );
void f() {
    compute( 0 );
}

```

Что произойдет в случае, если объявления `using` будут расположены в функции `main()` перед обращением к функции `compute()`? Ответьте на те же вопросы, что и в предыдущем упражнении.

18.3. Множественное и виртуальное наследование

Множественное наследование (multiple inheritance) — это способность получить класс как производный непосредственно от нескольких базовых классов (см. раздел 15.2.2). Полученный в результате класс наследует свойства всех своих базовых классов. Несмотря на простоту концепции, одновременное использование нескольких базовых классов может создать достаточно много сложностей как на этапе проектирования, так и на этапе реализации.

Для исследования множественного наследования используем пример иерархии из животного мира. Животные расположены на разных уровнях абстракции. Есть индивидуальные животные, различающиеся по именам, такие как Ling-ling (Линг-линг), Mowgli (Маугли) и Baloo (Балу). Каждое животное можно отнести к определенному виду; Линг-линг, например, это гигантская панда. Виды в свою очередь относятся к определенным семействам. Гигантская панда принадлежит к семейству медведей, а каждое семейство является членом сообщества животного мира.

Каждый уровень абстракции содержит разнообразные данные и функции. Определим класс ZooAnimal как абстрактный, призванный содержать информацию, которая является общей для всех животных и предоставляет открытый интерфейс. Класс Bear (Медведь) будет содержать информацию, которая является специфической для семейства медведей, и т.д.

Кроме классов животных, здесь можно определить дополнительные классы, которые инкапсулируют различные абстракции, например, животных, подвергающихся опасности. В данной реализации класс Panda (Панда) будет получен в результате множественного наследования от классов Bear и Endangered (Подвергающийся опасности).

18.3.1. Множественное наследование

Список наследования производного класса может содержать несколько базовых классов:

```
class Bear : public ZooAnimal { /* ... */ ;  
class Panda : public Bear, public Endangered { /*  
... */ ;
```

У каждого базового класса есть необязательный спецификатор доступа

(см. раздел 15.5). Как обычно, если спецификатор доступа отсутствует, по умолчанию подразумевается спецификатор `private` (закрытый), если используется ключевое слово `class`, и `public` (открытый), если используется ключевое слово `struct` (см. раздел 15.5).

Как и при одиночном наследовании, список наследования может включить только те классы, которые были определены и не были определены как `final` (см. раздел 15.2.2). Язык C++ не налагает никаких ограничений на количество базовых классов, из которых может быть получен производный класс. Однако базовый класс может присутствовать в списке наследования только один раз.

***При множественном наследовании классы наследуют
состояние каждого из базовых классов***

При множественном наследовании объект производного класса внутренне содержит объекты каждого из своих базовых классов (см. раздел 15.2.2). Например, на рис. 18.2 у объекта `Panda` есть часть класса `Bear` (которая сама содержит часть `ZooAnimal`), часть класса `Endangered` и нестатические переменные-члены, если таковые имеются, объявленные в пределах класса `Panda`.



Рис. 18.2. Концептуальная структура объекта класса Panda

***Конструкторы производного класса инициализируют все
объекты базовых классов***

Создание объекта производного класса подразумевает создание и инициализацию внутренних объектов всех его базовых классов. В случае одиночного наследования из (единого) базового класса (см. раздел 15.2.2) в списке инициализации конструктора производного класса можно передать значения только для прямых базовых классов:

```
// явная инициализация объектов обоих базовых
классов
Panda::Panda( std::string name, bool onExhibit )
: Bear( name, onExhibit, "Panda" ),
  Endangered( Endangered::critical ) { }
// неявное применение стандартного конструктора
класса Bear для
// инициализации его внутреннего объекта
Panda::Panda()
: Endangered( Endangered::critical ) { }
```

Список инициализации конструктора позволяет передать аргументы каждому из прямых базовых классов, однако порядок выполнения конструкторов (constructor order) зависит от порядка их расположения в списке наследования класса. Порядок их расположения в списке инициализации конструктора не имеет значения. Объект класса Panda инициализируется следующим образом.

- Внутренний объект класса ZooAnimal, самого первого базового класса иерархии класса Panda, непосредственного базового для класса Bear создается первым.
- Внутренний объект класса Bear, первого непосредственного базового класса для класса Panda, инициализируется следующим.
- Внутренний объект класса Endangered, второго непосредственного базового класса для класса Panda, инициализируется следующим.
- Последней инициализируется наиболее производная часть класса Panda.

Унаследованные конструкторы и множественное наследование



По новому стандарту производный класс может наследовать свои конструкторы от одного или нескольких своих базовых классов (см. раздел 15.7.4). Нельзя наследовать тот же конструктор (т.е. конструктор с тем же списком параметров) от более чем одного базового класса:

```

struct Base1 {
    Base1() = default;
    Base1( const std::string& );
    Base1( std::shared_ptr<int> );
};

struct Base2 {
    Base2() = default;
    Base2( const std::string& );
    Base2( int );
};

// ошибка: D1 пытается унаследовать D1::D1( const
string& ) от обоих
// базовых классов
struct D1: public Base1, public Base2 {
    using Base1::Base1; // наследует конструкторы от
Base1
    using Base2::Base2; // наследует конструкторы от
Base2
};

Класс, унаследовавший тот же конструктор от нескольких базовых
классов, должен определить собственную версию этого конструктора:
struct D2: public Base1, public Base2 {
    using Base1::Base1; // наследует конструкторы от
Base1
    using Base2::Base2; // наследует конструкторы от
Base2
    // D2 должен определить собственный конструктор,
получающий string
    D2( const string &s ) : Base1(s), Base2(s) { }
    D2() = default; // необходимо, поскольку D2
определяет собственный
                                // конструктор
};

```

Деструкторы и множественное наследование

Как обычно, деструктор в производном классе отвечает за освобождение ресурсов, зарезервированных этим классом. Автоматически освобождаются члены только производного класса и всех базовых классов. Тело синтезируемого деструктора пусто.

Деструкторы всегда выполняются в порядке, обратном вызову конструкторов. В данном примере порядок вызова деструкторов будет следующим: `~Panda()`, `~Endangered()`, `~Bear()`, `~ZooAnimal()`.

Функции копирования и перемещения при множественном наследовании

Как и в случае одиночного наследования, классы с несколькими базовыми классами, определяющими собственные конструкторы копирования, перемещения и операторы присвоения, должны копировать, перемещать и присваивать весь объект (см. раздел 15.7.2). Базовые части класса, производного от нескольких базовых, автоматически копируются, перемещаются и присваиваются, только если производный класс использует синтезируемые версии этих функций-членов. В синтезируемых функциях-членах управления копированием каждый базовый класс неявно создается, присваивается или удаляется с использованием соответствующего члена базового класса.

Например, если класс `Panda` использует синтезируемые функции-члены, то инициализация объекта `ling_ling` вызовет конструктор копий класса `Bear`, который в свою очередь вызовет конструктор копий класса `ZooAnimal` прежде, чем выполнить конструктор копий класса `Bear`:

```
Panda ying_yang("ying_yang");
Panda ling_ling = ying_yang; // использует
конструктор копий
```

Как только часть `Bear` объекта `ling_ling` создана, выполняется конструктор копий класса `Endangered`, создающий соответствующую часть объекта. И наконец, выполняется конструктор копий класса `Panda`. Аналогично для синтезируемого конструктора перемещения.

Синтезируемый оператор присвоения копии ведет себя так же, как и конструктор копий. Сначала он присваивает часть `Bear` (и его часть `ZooAnimal`) объекта, затем часть `Endangered` и наконец часть `Panda`. Оператор присвоения при перемещении ведет себя подобным образом.

Упражнения раздела 18.3.1

Упражнение 18.21. Объясните следующие объявления. Найдите все ошибки и объясните их причину:

- (a) class CADVehicle : public CAD, Vehicle { ... };
- (b) class DblList: public List, public List { ... }

```
(c) class iostream: public istream, public ostream  
{ ... };
```

Упражнение 18.22. С учетом следующей иерархии класса, в которой у каждого класса определен стандартный конструктор:

```
class A { ... };  
class B : public A { ... };  
class C : public B { ... };  
class X { ... };  
class Y { ... };  
class Z : public X, public Y { ... };  
class MI : public C, public Z { ... };
```

Каков порядок выполнения конструкторов при создании следующего объекта?

```
MI mi;
```

18.3.2. Преобразования и несколько базовых классов

При одиночном наследовании указатель или ссылка на производный класс могут быть автоматически преобразованы в указатель или ссылку на базовый класс (см. раздел 15.2.2 и раздел 15.5). Это справедливо и для множественного наследования. Указатель или ссылка на производный класс могут быть преобразованы в указатель или ссылку на любой из его базовых классов. Например, указатель или ссылка на класс ZooAnimal, Bear или Endangered может указывать или ссылаться на объект класса Panda.

// функции, получающие ссылки на класс, базовый для класса Panda

```
void print(const Bear&);  
void highlight(const Endangered&);  
ostream& operator<<(ostream&, const ZooAnimal&);  
Panda ying_yang("ying_yang");  
print(ying_yang); // передает объект  
класса Panda как // ссылку на объект  
класса Bear  
highlight(ying_yang); // передает объект  
класса Panda как // ссылку на объект  
класса Endangered
```

```
cout << ying_yang << endl; // передает объект
класса Panda как
// ссылку на объект
класса ZooAnimal
```

Компилятор даже не пытается как-то различать базовые классы. Преобразования в каждый из базовых классов происходят одинаково успешно. Рассмотрим, например, перегруженную версию функции `print()`:

```
void print( const Bear& );
void print( const Endangered& );
```

Вызов функции `print()` без квалификации для объекта класса `Panda` приведет к ошибке во время выполнения.

```
Panda ying_yang( "ying_yang" );
print( ying_yang ); // ошибка: неоднозначность
```

Поиск на основании типа указателя или ссылки

Как и при одиночном наследовании, статический тип объекта, указателя или ссылки определяет, какие из членов можно использовать. Если используется указатель класса `ZooAnimal`, для применения будут пригодны только те функции, которые определены в этом классе. Части интерфейса класса `Panda`, специфические для классов `Bear`, `Panda` и `Endangered`, окажутся недоступны. Аналогично указатель или ссылка на класс `Bear` применимы только для доступа к членам классов `Bear` и `ZooAnimal`, а указатель или ссылка на класс `Endangered` ограничены лишь членами класса `Endangered`.

В качестве примера рассмотрим следующие вызовы с учетом того, что эти классы определяют виртуальные функции, перечисленные в табл. 18.1.

```
Bear *pb = new Panda( "ying_yang" );
pb->print(); // ok: Panda::print()
pb->cuddle(); // ошибка: не является частью
интерфейса Bear
pb->highlight(); // ошибка: не является частью
интерфейса Bear
delete pb; // ok: Panda::~Panda()
```

Когда объект класса `Panda` используется при помощи указателя или ссылки на класс `Endangered`, части объекта класса `Panda`, специфические для классов `Panda` и `Bear`, становятся недоступными.

```
Endangered *pe = new Panda( "ying_yang" );
```

```

pe->print();           // ok: Panda:: print()
pe->toes();            // ошибка: не является частью
истерфейса Endangered
pe->cuddle();          // ошибка: не является частью
истерфейса Endangered
pe->highlight();       // ok: Panda:: highlight()
delete pe;              // ok: Panda:: ~Panda()

```

Таблица 18.1. Виртуальные функции иерархии классов ZooAnimal/Endangered

Функция	Класс, определяющий собственную версию
print()	ZooAnimal::ZooAnimal
	Bear:: Bear
	Endangered:: Endangered
	Panda:: Panda
highlight	Endangered:: Endangered
	Panda:: Panda
toes	Bear:: Bear
	Panda:: Panda
cuddle	Panda:: Panda
Деструктор	ZooAnimal:: ZooAnimal
	Endangered:: Endangered

Упражнения раздела 18.3.2

Упражнение 18.23. Используя иерархию из упражнения 18.22, а также определенный ниже класс D и с учетом наличия у каждого класса стандартного конструктора, укажите, какие из следующих преобразований недопустимы (если таковые вообще имеются)?

```

class D : public X, public C { ... };
D *pd = new D;
(a) X *px = pd; (b) A *pa = pd;
(c) B *pb = pd; (d) C *pc = pd;

```

Упражнение 18.24. Выше представлена последовательность вызовов через указатель на класс Bear, указывающих на объект класса Panda. Объясните каждый вызов, подразумевая, что вместо него используется указатель на класс ZooAnimal, указывающий на объект класса Panda.

Упражнение 18.25. Предположим, существуют два базовых класса,

Base1 и Base2, в каждом из которых определена виртуальная функция-член по имени print() и виртуальный деструктор. От этих базовых классов были получены следующие классы, в каждом из которых переопределена функция print().

```
class D1 : public Base1 { /* ... */ };
class D2 : public Base2 { /* ... */ };
class MI : public D1, public D2 { /* ... */ };
```

Используя следующие определения, укажите, какая из функций используется при каждом вызове:

```
Base1 *pb1 = new MI;
Base2 *pb2 = new MI;
D1 *pd1 = new MI;
D2 *pd2 = new MI;
(a) pb1->print(); (b) pd1->print(); (c) pd2-
>print();
(d) delete pb2; (e) delete pd1; (f) delete pd2;
```

18.3.3. Область видимости класса при множественном наследовании

При одиночном наследовании область видимости производного класса вкладывается в пределы его прямых и косвенных базовых классов (см. раздел 15.6). Поиск имен осуществляется по всей иерархии наследования. Имена, определенные в производном классе, скрывают совпадающие имена в базовом классе.

При множественном наследовании поиск осуществляется *одновременно* во всех прямых базовых классах. Если имя находится в нескольких базовых классах, происходит ошибка неоднозначности.

В рассматриваемом примере, если имя используется через указатель, ссылку или объект класса Panda, дерева иерархии Endangered и Bear/ZooAnimal исследуются параллельно. Если имя находится в нескольких иерархиях, то возникнет неоднозначность. Для класса вполне допустимо наследовать несколько членов с тем же именем. Но если это имя необходимо использовать, следует указать, какая именно версия имеется в виду.



Когда у класса есть несколько базовых классов, производный класс вполне может унаследовать одноименный член от двух и более своих базовых классов. При использовании этого имени без уточнения класса происходит неоднозначность.

Например, если классы ZooAnimal и Endangered определяют функцию-член max_weight(), а класс Panda не определяет ее, то следующий вызов ошибочен:

```
double d = ying_yang.max_weight();
```

В результате наследования класс Panda получает две функции-члена max_weight(), что совершенно допустимо. Наследование создает потенциальную неоднозначность. Ее вполне можно избежать, если объект Panda не будет вызывать функцию-член max_weight(). Ошибки также можно избежать, если явно указать требуемую версию функции: ZooAnimal::max_weight() или Endangered::max_weight(). Ошибка неоднозначности произойдет только при попытке использования функции без уточнения.

Неоднозначность двойного наследования функции-члена max_weight вполне очевидна и логична. Удивительно узнать то, что ошибка произошла бы, даже если у двух наследованных функций были разные списки параметров. Точно так же эта ошибка произошла бы даже в случае, если бы функция max_weight() была закрытой в одном классе и открытой или защищенной в другом. И наконец, если бы функция max_weight() была определена в классе Bear, а не в классе ZooAnimal, то вызов все равно был бы ошибочен.

Как обычно, поиск имени осуществляется под контролем соответствия типов (см. раздел 6.4.1). Когда компилятор находит имя функции max_weight() в двух разных областях видимости, он оповещает об ошибке неоднозначности.

Проще всего избежать потенциальных неоднозначностей, определив версию такой функции в производном классе. Например, снабдив класс Panda функцией max_weight(), можно решить все проблемы:

```
double Panda::max_weight() const {
    return std::max(ZooAnimal::max_weight(),
                    Endangered::max_weight());
}
```

Упражнения раздела 18.3.3

Упражнение 18.26. С учетом иерархии кода для упражнений объясните, почему ошибочен следующий вызов функции `print()`? Исправьте структуру М1 так, чтобы позволить этот вызов.

```
M1 mi;
mi.print(42);
```

Упражнение 18.27. С учетом иерархии кода для упражнений и того, что в структуру М1 добавлена приведенная ниже функция `foo()`, ответьте на следующие вопросы:

```
int ival;
double dval;
void M1::foo(double cval) {
    int dval;
    // варианты вопросов упражнения располагаются
здесь ...
}
```

- (a) Перечислите все имена, видимые из функции `M1::foo()`.
- (b) Видимы ли какие-нибудь имена из больше чем одного базового класса?
- (c) Присвойте локальному экземпляру переменной `dval` сумму переменных-членов `dval` объектов классов `Base1` и `Derived`.
- (d) Присвойте значение последнего элемента вектора `M1::dvec` переменной-члену `Base2::fval`.
- (e) Присвойте переменной-члену `cval` класса `Base1` первый символ строки `sval` класса `Derived`.

Код для упражнений раздела 18.3.3

```
struct Base1 {
    void print(int) const; // по умолчанию открыты
protected:
    int ival;
    double dval;
    char cval;
private:
    int *id;
};

struct Base2 {
    void print(double) const; // по умолчанию открыты
protected:
```

```

        double fval;
private:
    double dval;
};

struct Derived : public Basel {
    void print(std::string) const; // по умолчанию
открыты
protected:
    std::string sval;
    double dval;
};

struct MI : public Derived, public Base2 {
    void print(std::vector<double>); // по умолчанию
открыты
protected:
    int *ival;
    std::vector<double> dvec;
};

```

18.3.4. Виртуальное наследование

Хотя список наследования класса не может включать тот же базовый класс несколько раз, класс вполне может унаследовать тот же базовый класс многократно. Тот же базовый класс может быть унаследован косвенно, от двух его собственных прямых базовых классов, либо он может унаследовать некий класс и прямо, и косвенно, через другой из его базовых классов.

Например, библиотечные классы ввода-вывода `istream` и `ostream` происходят от общего абстрактного базового класса `basic_ios`. Этот класс содержит буфер потока и управляет флагом состояния потока. Класс `iostream`, способный и читать, и писать в поток, происходит непосредственно и от класса `istream`, и от класса `ostream`. Поскольку оба класса происходят от класса `basic_ios`, класс `iostream` наследует этот базовый класс дважды: один раз от класса `istream` и один раз от класса `ostream`.

По умолчанию объект производного класса содержит отдельные части, соответствующие каждому классу в его цепи наследования. Если тот же базовый класс наследуется несколько раз, то у объекта производного класса будет больше одного внутреннего объекта этого типа.

Для такого класса, как `iostream`, это стандартное поведение не работает. Объект класса `iostream` должен использовать тот же буфер и для чтения, и для записи, а его флаг должен отражать состояние операций и ввода, и вывода. Если у объекта класса `iostream` будут две копии объекта класса `basic_ios`, то их совместное использование невозможно.

В языке C++ для решения этой проблемы используется *виртуальное наследование* (virtual inheritance). Виртуальное наследование позволяет классу указать, что его базовый класс будет использоваться совместно. Совместно используемый внутренний объект базового класса называется *виртуальным базовым классом* (virtual base class). Независимо от того, сколько раз тот же базовый виртуальный класс присутствует в иерархии наследования, объект производного класса содержит только один совместно используемый внутренний объект этого виртуального базового класса.

Разные классы Panda

В прошлом велись дебаты о принадлежности вида панда к семейству енотов или медведей. Чтобы отобразить эти сомнения, изменим класс `Panda` так, чтобы он происходил и от класса `Bear`, и от класса `Raccoon`. Чтобы избавить класс `Panda` от двух частей базового класса `ZooAnimal`, определим наследование классов `Bear` и `Raccoon` от класса `ZooAnimal` как виртуальное. Новая иерархия представлена на рис. 18.3.

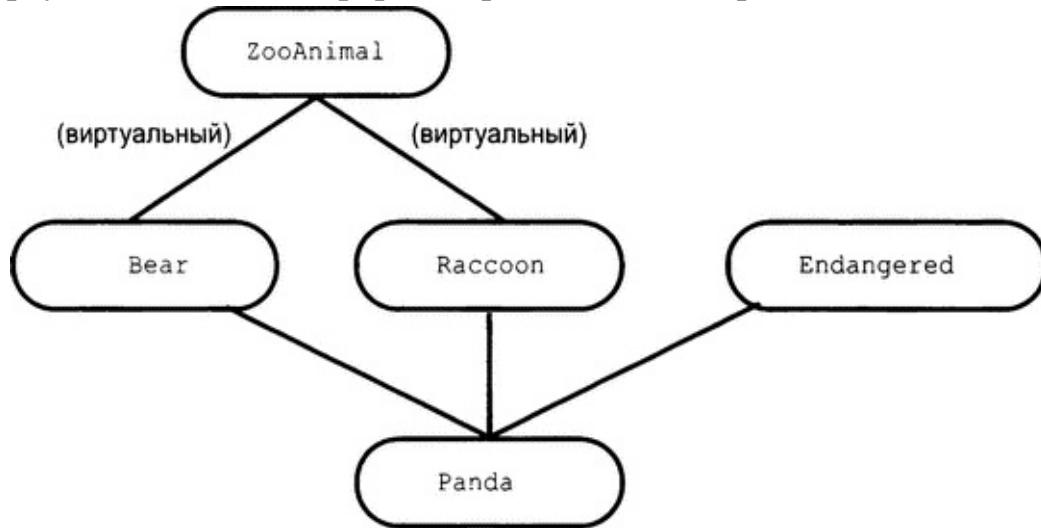


Рис. 18.3. Виртуальное наследование в иерархии класса `Panda`

Глядя на новую иерархию, можно заметить неочевидный аспект виртуального наследования. Виртуальное наследование должно быть

осуществлено прежде, чем в нем возникнет потребность. Например, в этих классах потребность в виртуальном наследовании возникает только при определении класса Panda. Но если бы классы Bear и Raccoon не определили бы свое происхождение от класса ZooAnimal как виртуальное, конструкция класса Panda была бы неудачна.

На практике необходимость наличия промежуточного базового класса при виртуальном наследовании редко создает проблемы. Обычно иерархия классов, в которой используется виртуальное наследование, разрабатывается сразу и одним лицом (или группой разработчиков). Ситуации, когда разработку виртуального базового класса необходимо поручить независимому производителю, чрезвычайно редки, а разработчик нового базового класса не может внести изменения в существующую иерархию.



Виртуальное наследование влияет на те классы, которые происходят от виртуального базового класса впоследствии; оно не влияет на класс производный непосредственно.

Использование виртуального базового класса

Базовый класс объявляется виртуальным при помощи ключевого слова `virtual` в списке наследования:

```
// порядок расположения ключевых слов public и
virtual несуществен
class Raccoon : public virtual ZooAnimal { /* ... */
};
class Bear : virtual public ZooAnimal { /* ... */ }
```

Здесь класс ZooAnimal объявлен виртуальным базовым для классов Bear и Raccoon.

Спецификатор `virtual` заявляет о готовности совместно использовать единый экземпляр указанного базового класса в последующих производных классах. Нет никаких особых ограничителей на классы, используемые как виртуальные базовые классы.

Для наследования от класса, имеющего виртуальный базовый класс, не нужно ничего особенного:

```
class Panda : public Bear,  
                  public Raccoon, public Endangered {  
};
```

Здесь класс Panda наследует класс ZooAnimal через два своих базовых класса — Raccoon и Bear. Но поскольку эти классы происходят от класса ZooAnimal виртуально, у класса Panda есть только одна часть базового класса ZooAnimal.

Для базовых классов поддерживаются стандартные преобразования

Объектом производного класса можно манипулировать как обычно, при помощи указателя или ссылки на базовый класс, хотя он и является виртуальным. Например, все следующие преобразования для базового класса объекта класса Panda вполне допустимы:

```
void dance( const Bear& );  
void rummage( const Raccoon& );  
ostream& operator<<( ostream&, const ZooAnimal& );  
Panda ying_yang;  
dance( ying_yang );      // ok: передает объект Panda  
как Bear  
rummage( ying_yang );   // ok: передает объект Panda  
как Raccoon  
cout << ying_yang;     // ok: передает объект Panda  
как ZooAnimal
```

Видимость членов виртуальных базовых классов

Поскольку виртуальному базовому классу соответствует только один совместно используемый внутренний объект, к членам объекта этого базового класса можно обратиться непосредственно и однозначно. Кроме того, если член виртуального базового класса переопределяется только в одной ветви наследования, к этому переопределенному члену класса можно обратиться непосредственно. Если член переопределяется больше чем одним базовым классом, то производный класс вообще должен определить собственную версию этого члена.

Предположим, например, что класс B определяет члены по имени x; класс D1 виртуально наследует класс B, как и класс D2; а класс D происходит от классов D1 и D2. Из области видимости класса D член x видим через оба своих базовых класса. Есть три возможности использовать член x через объект класса D:

- Если член *x* не будет определен ни в классе D1, ни в D2, то будет использован член класса B; никакой неоднозначности нет. Объект класса D содержит только один экземпляр члена *x*.
- Если *x* является членом класса B и одного (но не обоих) из классов D1 или D2, никакой неоднозначности снова нет: версия в производном классе имеет приоритет перед совместно используемым виртуальным базовым классом B.
- Если член *x* определяется и в классе D1, и в классе D2, то прямой доступ к этому члену неоднозначен.

Как и в иерархии с невиртуальным множественным наследованием, подобная неоднозначность лучше всего устраняется переопределением члена в производном классе.

Упражнения раздела 18.3.4

Упражнение 18.28. Рассмотрим следующую иерархию класса. Можно ли в классе VMI обращаться к унаследованным членам без уточнения? Какие из них требуют полностью квалифицированных имен? Объясните, почему.

```
struct Base {
    void bar( int ); // по умолчанию открыты
protected:
    int ival;
};

struct Derived1 : virtual public Base {
    void bar( char ); // по умолчанию открыты
    void foo( char );
protected:
    char cval;
};

struct Derived2 : virtual public Base {
    void foo( int ); // по умолчанию открыты
protected:
    int ival;
    char cval;
};

class VMI : public Derived1, public Derived2 { };
```

18.3.5. Конструкторы и виртуальное наследование

При виртуальном наследовании виртуальный базовый класс инициализируется *конструктором самого последнего производного класса*. В рассматриваемом примере при создании объекта класса Panda инициализацию членов базового класса ZooAnimal контролирует конструктор класса Panda.

Чтобы понять это правило, рассмотрим происходящее при применении обычных правил инициализации. В этом случае объект виртуального базового класса мог бы быть инициализирован несколько раз. Он был бы инициализирован вдоль каждой ветви наследования, содержащей этот виртуальный базовый класс. В данном примере, если бы к классу ZooAnimal применялись обычные правила инициализации, то части Bear и Raccoon инициализировали бы часть ZooAnimal объекта класса Panda.

Конечно, каждый базовый класс в иерархии объекта мог бы в некоторый момент быть "более производным". Поскольку вполне можно создавать независимые объекты класса, производного от виртуального базового класса, конструкторы в этом классе должны инициализировать его виртуальный базовый класс. Например, когда в рассматриваемой иерархии создается объект класса Bear (или Raccoon), никакого дальнейшего применения производного класса нет. В данном случае конструкторы класса Bear (или Raccoon) непосредственно инициализируют базовую часть ZooAnimal, как обычно:

```
Bear:: Bear( std::string name, bool onExhibit ) :  
    ZooAnimal( name, onExhibit, "Bear" ) { }  
Raccoon:: Raccoon( std::string name, bool onExhibit ) :  
    ZooAnimal( name, onExhibit, "Raccoon" ) { }
```

Когда создается объект класса Panda, он является наиболее производным типом и контролирует инициализацию совместно используемого базового класса ZooAnimal. Даже при том, что класс ZooAnimal не является прямым базовым классом для класса Panda, часть ZooAnimal инициализирует конструктор класса Panda:

```
Panda:: Panda( std::string name, bool onExhibit )  
    : ZooAnimal( name, onExhibit, "Panda" ),  
        Bear( name, onExhibit ),  
        Raccoon( name, onExhibit ),  
        Endangered( Endangered:: critical ),  
        sleeping_flag( false ) { }
```

Как создается объект при виртуальном наследовании

Порядок создания объекта с виртуальным базовым классом немного отличается от обычного: сначала инициализируется часть виртуального базового класса с использованием инициализаторов, предоставленных в конструкторе для наиболее производного класса. Как только создана часть виртуального базового класса, создаются части прямых базовых классов в порядке их расположения в списке наследования.

Например, объект класса Panda создается так.

- Сначала создается часть виртуального базового класса ZooAnimal. При этом используются инициализаторы из списка инициализации конструктора класса Panda.

- Затем создается часть Bear.
- Затем создается часть Raccoon.
- Следующей создается часть прямого базового класса Endangered.
- Наконец создается часть Panda.

Если конструктор класса Panda не инициализирует явно часть базового класса ZooAnimal, будет использован стандартный конструктор класса ZooAnimal. Если у класса ZooAnimal нет стандартного конструктора, произойдет ошибка.



Части виртуальных базовых классов всегда создаются до частей обычных базовых классов, независимо от того, где они располагаются в иерархии наследования.

Порядок выполнения конструкторов и деструкторов

У класса может быть несколько виртуальных базовых классов. В этом случае части виртуальных классов создаются в порядке их расположения в списке наследования. Например, в следующей иерархии наследования у класса TeddyBear (МедвежонокТедди) есть два виртуальных базовых класса: прямой виртуальный базовый класс ToyAnimal (ИгрушечноеЖивотное) и косвенный базовый класс ZooAnimal, от которого происходит класс Bear:

```
class Character { /* ... */ };
class BookCharacter : public Character { /* ... */ }
```

```

};

class ToyAnimal { /* ... */ };
class TeddyBear : public BookCharacter,
public Bear, public virtual ToyAnimal
{ /* ... */ };

```

Чтобы выявить наличие виртуальных базовых классов, прямые базовые классы просматриваются в порядке объявления. Если это так, то сначала создаются части виртуальных базовых классов, затем выполняются конструкторы обычных, не виртуальных базовых классов в порядке их объявления. Таким образом, чтобы создать объект класса `TeddyBear`, конструкторы его частей вызываются в следующем порядке:

```

ZooAnimal();           // виртуальный базовый класс Bear
ToyAnimal();           // прямой виртуальный базовый
класс
Character();          // косвенный базовый класс первого
не виртуального
                       // базового класса
BookCharacter();      // первый прямой не виртуальный
базовый класс
Bear();                // второй прямой не виртуальный
базовый класс
TeddyBear();           // наиболее производный класс

```

Тот же порядок создания используется в синтезируемом конструкторе копий и конструкторах перемещения, в синтезируемых операторах присвоения члены присваиваются в том же порядке. Вызов деструкторов базовых классов осуществляется в порядке, обратном порядку вызова конструкторов. Часть `TeddyBear` будет удалена сначала, а часть `ZooAnimal` — последней.

Упражнения раздела 18.3.5

Упражнение 18.29. Имеется следующая иерархия классов:

```

class Class { ... };

class Base : public Class { ... };

class D1 : virtual public Base { ... };

class D2 : virtual public Base { ... };

class MI : public D1, public D2 { ... };

class Final : public MI, public Class { ... };

```

(a) Каков порядок вызова конструкторов и деструкторов объектов класса `Final`?

(b) Сколько внутренних объектов класса `Base` находится в объекте класса `Final`? А сколько внутренних объектов класса `Class`?

(c) Какие из следующих случаев присвоения приведут к ошибке во время компиляции?

```
Base *pb; Class *pc; MI *pmi; D2 *pd2;  
( a) pb = new Class; ( b) pc = new Final;  
( c) pmi = pb; ( d) pd2 = pmi;
```

Упражнение 18.30. Определите в классе `Base` стандартный конструктор, конструктор копий и конструктор с параметром типа `int`. Определите те же три конструктора в каждом производном классе. Каждый конструктор должен использовать свой аргумент для инициализации своей части `Base`.

Резюме

Язык C++ применяется для решения широкого диапазона проблем: от требующих лишь нескольких часов работы до занимающих годы работы больших групп разработчиков. Некоторые из средств языка C++ наиболее полезны при создании крупномасштабных приложений. Имеется в виду обработка исключений, пространства имен и множественное или виртуальное наследование.

Обработка исключений позволяет отделить ту часть кода, где может произойти ошибка, от той части кода, где она обрабатывается. При передаче исключения выполнение текущей функции приостанавливается и начинается поиск ближайшей директивы `catch`. Локальные переменные, определенные в покидаемых при поиске директив `catch` функциях, удаляются в ходе обработки исключения.

Пространства имен — это механизм управления большими и сложными приложениями, формируемыми из кода, созданного независимыми поставщиками. Пространство имен является областью видимости, в которой могут быть определены объекты, типы, функции, шаблоны и другие пространства имен. Стандартная библиотека определена в пространстве имен `std`.

С концептуальной точки зрения множественное наследование — довольно простое понятие: производный класс может быть унаследован от нескольких прямых базовых классов. Объект производного класса состоит из частей, представляющих собой внутренние объекты всех своих базовых классов. Концепция действительно проста, но на практике сопряжена со многими сложностями. В частности, наследование от нескольких базовых

классов создает вероятность конфликтов имен и в результате порождает неоднозначные обращения к именам из базовых частей объекта.

Если класс происходит от нескольких непосредственных базовых классов, не исключена ситуация, когда эти классы сами могут иметь общий базовый класс. В таких случаях промежуточные классы могут применить виртуальное наследование, позволяющее другим классам иерархии, унаследовавшим тот же базовый класс, совместно использовать его внутренний объект. Таким образом, объект производного класса будет иметь только одну совместно используемую копию внутреннего объекта виртуального базового класса.

Термины

Безымянное пространство имен (unnamed namespace). Пространство имен, определенное без имени. К именам, определенным в безымянном пространстве имен, можно обращаться непосредственно, без оператора области видимости. Каждый файл имеет собственное, уникальное безымянное пространство имен. Имена в файле невидимы вне данного файла.

Блок try (try block). Блок операторов, начинающийся ключевым словом `try` и содержащий одну или несколько директив `catch`. Если код в блоке `try` передает исключение и одна из директив `catch` соответствует типу переданного исключения, то переданное исключение будет обработано этим обработчиком. В противном случае исключение будет передано из блока `try` другому обработчику, далее по цепи вызовов.

Блок try функции (function try block). Используется для обработки исключений из списка инициализации конструктора. Ключевое слово `try` располагается перед двоеточием, начинающим список инициализации конструктора (или перед открывающей фигурной скобкой тела конструктора, если список инициализации пуст), и завершается одной или несколькими директивами `catch`, которые следуют после закрывающей фигурной скобки тела конструктора.

Виртуальное наследование (virtual inheritance). Форма множественного наследования, при котором производные классы совместно используют одну копию экземпляра базового класса, даже если в иерархии он встречается несколько раз.

Виртуальный базовый класс (virtual base class). Базовый класс, при наследовании которого было использовано ключевое слово `virtual`. В объекте производного класса часть виртуального базового класса

содержится только в одном экземпляре, даже если в иерархии этот класс присутствует несколько раз. При не виртуальном наследовании конструктор может инициализировать только непосредственный базовый класс (классы). При виртуальном наследовании этот класс мог бы быть инициализирован несколькими производными классами, которые должны предоставить инициализирующие значения для всех его виртуальных предков.

Выражениет`throw` е (передача исключения). Выражение, которое прерывает текущий поток выполнения. Каждый оператор `throw` передает управление ближайшему окружающему блоку `catch`, который способен обработать исключение переданного типа. Выражение `e` будет скопировано в объект исключения.

Глобальное пространство имен (global namespace). Неявное пространство имен, содержащее все определения глобальных объектов, которыми обладает каждая программа.

Директива`catch` (`catch` clause). Часть программы, которая обрабатывает исключение. Директива обработчика состоит из ключевого слова `catch`, за которым следуют объявление исключения и блок операторов. Код в блоке `catch` предназначен для обработки исключений типа, указанного в его объявлении.

Директива`using` (`using` directive). Объявление в форме `using NS;` делает *все* имена пространства имен `NS` доступными в ближайшей области видимости, содержащей и директиву `using`, и само пространство имен.

Загромождение пространства имен (namespace pollution). Термин, используемый для описания ситуации, когда все имена классов и функций располагаются в глобальном пространстве имен. Большие программы, использующие код, который создан несколькими независимыми производителями, зачастую сталкиваются с конфликтами имен, если эти имена глобальны.

Множественное наследование (multiple inheritance). Наследование, при котором класс имеет несколько непосредственных базовых классов. Производный класс наследует члены всех своих базовых классов. Имена нескольких базовых классов указываются в списке наследования класса. Для каждого базового класса может быть предоставлен отдельный спецификатор доступа.

Обработка исключений (exception handling). Механизм уровня языка, предназначенный для ликвидации аномалий времени выполнения. Один независимо разработанный раздел кода может обнаружить проблему и

передать исключение, которое может получить и обработать другая независимо разработанная часть программы. Часть кода, обнаруживающая ошибку, передает исключение, а часть кода, получающая его, осуществляет обработку.

Обработчик (handler). Синоним директивы `catch`.

Обработчик для всех исключений (catch-all). Директива `catch`, в которой объявляется исключение. Директива обработчика для всех исключений обрабатывает исключения любого типа. Обычно он используется для предварительной обработки исключения, осуществляющей локально. Затем исключение повторно передается другой части программы, в которой и осуществляется устранение причины проблем.

Объект исключения (exception object). Объект, используемый для передачи сообщения между блоками `throw` и `catch`. Объект создается в точке передачи и является копией использованного выражения. Объект исключения существует, пока не сработает последний обработчик для его типа. Тип объекта соответствует типу использованного выражения.

Объявление using (using declaration). Механизм, позволяющий ввести одно имя из пространства имен в текущую область видимости. `using std::cout;`. Это объявление сделает имя `cout` из пространства имен `std` доступным в текущей области видимости, благодаря чему имя `cout` можно применять без спецификатора `std::`.

Объявление исключения (exception declaration). Объявление директивы `catch`, определяющее тип обрабатываемого исключения. Объявление действует как список параметров, каждый параметр которого инициализируется объектом исключения. Если спецификатор исключения имеет не ссылочный тип, то объект исключения копируется в обработчик.

Оператор noexcept. Оператор, возвращающий тип `bool` и указывающий, способно ли данное выражение передать исключение. Выражение не вычисляется. Результат — константное выражение. Его значение `true`, если выражение не содержит оператора `throw` и вызывает только те функции, которые не передают исключений; в противном случае результат — `false`.

Оператор области видимости (scope operator). Оператор `(::)` используется для доступа к именам пространства имен или класса.

Передача (raise). Синоним термина "throw" (передача). Программисты C++ используют термины "throwing" и "raising" как синонимы, означающие передачу исключения.

Повторная передача исключения (rethrow). Пустой оператор `throw` повторно передает объект исключения. Повторная передача возможна только из блока `catch` (обработчика) или из функции, прямо или косвенно вызываемой обработчиком. В результате будет повторно передан полученный ранее объект исключения.

Порядок выполнения конструкторов (constructor order). При не виртуальном наследовании части базовых классов строятся в том порядке, в котором они указаны в списке наследования класса. При виртуальном наследовании часть виртуального базового класса (классов) создается прежде любых других базовых классов. Они создаются в порядке расположения в списке наследования производного класса. Только самый последний производный тип может инициализировать виртуальный базовый класс; списки инициализации конструктора этого базового класса, расположенные в промежуточных базовых классах, игнорируются.

Прокрутка стека (stack unwinding). Процесс выхода из функции при передаче исключения и перехода к поиску его обработчика. Локальные объекты, созданные перед передачей исключения, удаляются перед началом поиска соответствующего обработчика.

Пространство имен (namespace). Механизм, используемый для сбора всех имен, определенных в библиотеке или другом фрагменте программы, в единую область видимости. В отличие от других областей видимости языка C++, область видимости пространства имен может быть определена в нескольких частях. Пространство имен может быть открыто, закрыто и открыто вновь, причем в разных частях программы.

Псевдоним пространства имен (namespace alias). Синтаксис создания синонима для пространства имен имеет следующий вид: `namespace N1 = N;` где `N1` — это лишь другое имя пространства имен `N`. Пространство имен может иметь несколько псевдонимов, причем псевдонимы и реальное имя пространства имен могут использоваться попарно.

Спецификация `noexcept`. Ключевое слово, обычно указывающее, передает ли функция исключение. Когда за списком параметров функции следует ключевое слово `noexcept`, за ним (необязательно) может следовать заключенное в скобки константное выражение, приводимое к типу `bool`. Если выражение отсутствует или возвращает значение `true`, функция не передает исключений. Если выражение возвращает значение `false` или у функции нет спецификации исключения, она может передать любое исключение.

Спецификация запрета передачи исключения (nonthrowing)

specification). Спецификация исключения, обещающая, что функция не будет передавать исключений. Если такая функция передаст исключение, то будет вызвана функция `terminate()`. К спецификаторам запрета передачи исключения относятся спецификатор `noexcept` без аргумента или с аргументом, возвращающим значение `true`, а также `throw()`.

Статический файловый объект (`file static`). Локальное для файла имя, которое было объявлено с использованием ключевого слова `static`. В языке C и версиях языка C++, выпущенных до появления стандарта, статические файловые объекты использовались для объявления таких объектов, которые применимы только в одном файле. Применение статических файловых объектов осуждено стандартом C++. Сейчас они заменены безымянными пространствами имен.

Функция `terminate()`. Библиотечная функция, вызов которой происходит в случае, когда переданное исключение либо так и не обработано, либо если оно было передано в обработчике исключений. Функция `terminate()` завершает выполнение программы.

Глава 19

Специализированные инструменты и технологии

В первых трех частях этой книги обсуждались аспекты языка C++, используемые практически всеми программистами C++. Кроме того, язык C++ предоставляет некоторые специализированные средства, которые большинство программистов используют крайне редко или не используют вообще.

Язык C++ предназначен для создания самых разнообразных приложений. В результате он обладает средствами, ненужными для одних приложений и иногда используемыми в других. В этой главе рассматриваются довольно редко используемые средства языка C++.

19.1. Контроль распределения памяти

Некоторые приложения нуждаются в специализированном распределении памяти, которое не могут обеспечить стандартные средства управления памятью. Разработчики таких приложений вынуждены вникать в подробности резервирования памяти, например, применения оператора `new` для помещения объекта в специфические виды памяти. Для этого они могут перегрузить операторы `new` и `delete` так, чтобы самостоятельно контролировать распределение памяти.

19.1.1. Перегрузка операторов `new` и `delete`

Хотя говорят, что можно "перегрузить операторы `new` и `delete`", перегрузка этих операторов весьма отличается от способа перегрузки других операторов. Чтобы понять, как их можно перегрузить, следует сначала узнать больше о том, как работают выражения `new` и `delete`.

Выражение `new` используется так:

```
// выражение new
string *sp = new string("a value"); // зарезервировать и
// инициализиро
сторку
string *arr = new string[10]; // зарезервировать
десять строк,
// инициализированных
значением по
// умолчанию
```

Фактически здесь три этапа: сначала выражение вызывает библиотечную функцию `operator new()` (или `operator new[]()`). Эта функция резервирует не типизированную область памяти достаточного размера для содержания объекта (или массива объектов) определенного типа. Затем компилятор запускает соответствующий конструктор, чтобы создать объект (объекты) из переданных инициализаторов. И наконец, возвращается указатель на вновь зарезервированный и созданный объект.

Выражение `delete` применяется для удаления динамически созданного объекта:

```
delete sp; // удалить *sp и освободить память,
// на которую указывает sp
delete [] arr; // удалить элементы массива и
освободить память
```

Здесь два этапа: сначала для объекта, на который указывает указатель `sp`, или для элементов массива, на который указывает имя `arr`, выполняется соответствующий деструктор. Затем компилятор освобождает память, вызвав библиотечную функцию `operator delete()` или `operator delete[]()` соответственно.

Приложения, которые собираются самостоятельно контролировать распределение памяти, определяют собственные версии функций `operator new()` и `operator delete()`. Даже при том, что

библиотека содержит определения этих функций, вполне можно определить их собственные версии, и компилятор не пожалуется на двойное определение. Вместо этого компилятор использует пользовательскую версию, а не определенную библиотекой.



ВНИМАНИЕ

При определении глобальных функций `operator new()` и `operator delete()` вся ответственность за динамическое распределение памяти ложится на разработчика. Эти функции *должны* быть корректны, так как являются жизненно важной частью всей программы.

Функции `operator new()` и `operator delete()` можно определить в глобальной области видимости или как функции-члены. Когда компилятор встречает выражение `new` или `delete`, он ищет соответствующую вызову функцию оператора. Если резервируемый (освобождаемый) объект имеет тип класса, то компилятор ищет сначала в пределах класса, включая все его базовые классы. Если у класса есть функции-члены `operator new()` и `operator delete()`, эти функции и используются в выражении `new` или `delete`. В противном случае компилятор ищет соответствующую функцию в глобальной области видимости. Если компилятор находит пользовательскую версию функции, он ее и использует для выполнения выражения `new` или `delete`. В противном случае используется версия из стандартной библиотеки.

Чтобы заставить выражение `new` или `delete` обойти функцию, предоставленную классом, и использовать таковую из глобальной области видимости, можно использовать оператор области видимости. Например, выражение `::new` имеет в виду функцию `operator new()` только из глобальной области видимости. Аналогично для выражения `::delete`.

Интерфейс функций `operator new()` и `operator delete()`

Библиотека определяет восемь перегруженных версий функций `operator new()` и `operator delete()`. Первые четыре версии оператора `new` способны передавать исключение `bad_alloc`. Следующие четыре версии оператора `new` не передают исключений:

```
// версии, способные передавать исключения
void *operator new( size_t ); //
```

резервирует объект
void *operator new[](size_t); //
резервирует массив
void *operator delete(void*) noexcept; //
освобождает объект
void *operator delete[](void*) noexcept; //
освобождает массив

// версии, обещающие не передавать исключений; см.
p. 12.1.2

void *operator new(size_t, noexcept_t&) noexcept;
void *operator new[](size_t, noexcept_t&) noexcept;
void *operator delete(void*, noexcept_t&) noexcept;
void *operator delete[](void*, noexcept_t&) noexcept;

Тип `nothrow_t` является структурой, определенной в заголовке `new`. У этого типа нет никаких членов. Заголовок `new` определяет также константный объект `nothrow`, который пользователи могут передавать как сигнал, что необходима версия оператора `new`, не передающего исключения (см. раздел 12.1.2). Будучи деструктором, функция `operator delete()` не должна передавать исключения (см. раздел 18.1.1). При перегрузке этих операторов следует определить, будут ли они передавать исключения. Для этого используется спецификатор исключения `noexcept` (см. раздел 18.1.4).

Приложение может определить свою собственную версию любой из этих функций. Если это так, то следует определить эти функции в глобальной области видимости или как функцию-член класса. Когда эти функции операторов определены как члены класса, они неявно являются статическими (см. раздел 7.6). Нет никакой необходимости объявлять их статическими явно, хотя сделать это вполне допустимо. Функции-члены операторов `new` и `delete` должны быть статическими, поскольку они используются до создания объекта (`operator new`) или после его удаления (`operator delete`). Поэтому у них нет никаких переменных-членов, которыми они могли бы манипулировать.

У функций `operator new()` и `operator new[]()` должен быть тип возвращаемого значения `void*`, а их первый параметр должен иметь тип `size_t`. У этого параметра не может быть аргумента по умолчанию. Функция `operator new()` используется при резервировании объекта;

функция `operator new[]()` вызывается при резервировании массива. Когда компилятор вызывает функцию `operator new()`, он инициализирует параметр типа `size_t` количеством байтов, необходимых для содержания объекта заданного типа; при вызове функции `operator new[]()` передается количество байтов, необходимых для хранения массива заданного количества элементов.

При определении собственной версии функции `operator new()` можно определить дополнительные параметры. Чтобы использующие такие функции выражения `new` могли передать аргументы этим дополнительным параметрам, следует применять размещающую форму оператора `new` (см. раздел 12.1.2). Хотя обычно вполне можно определить собственную версию функции `operator new()`, чтобы получить необходимый набор параметров, нельзя определить эту функцию в следующей форме:

```
void *operator new(size_t, void*); // эта версия не
может быть
```

//

переопределена

Данная конкретная форма зарезервирована для использования библиотекой и не может быть переопределена.

У функций `operator delete()` и `operator delete[]()` должен быть тип возвращаемого значения `void` и первый параметр типа `void*`. Выполнение выражения `delete` вызывает соответствующую функцию оператора и инициализирует ее параметр типа `void*` указателем на область памяти, подлежащую освобождению.

Когда функции `operator delete()` и `operator delete[]()` определяются как члены класса, у них может быть второй параметр типа `size_t`. Этот дополнительный параметр инициализируется размером (в байтах) объекта, заданного первым параметром. Параметр типа `size_t` используется при удалении объектов, являющихся частью иерархии наследования. Если у базового класса есть виртуальный деструктор (см. раздел 15.7.1), то передаваемый функции `operator delete()` размер зависит от динамического типа объекта, на который указывает удаляемый указатель. Кроме того, выполняемая версия функции `operator delete()` также будет зависеть от динамического типа объекта.

Терминология. Выражение `new` или функция `operator new()`

Имена библиотечных функций operator new() и operator delete() могут ввести в заблуждение. В отличие от других функций операторов (таких как operator=), эти функции не перегружают операторы new и delete. Фактически переопределить поведение операторов new и delete нельзя.

В процессе выполнения оператор new вызывает функцию operator new(), чтобы зарезервировать область памяти, в которой он затем создает объект. Оператор delete удаляет объект, а затем вызывает функцию operator delete(), чтобы освободить использованную объектом память.

Функции malloc() и free()

Если определяются собственные глобальные функции operator new() и operator delete(), они должны резервировать и освобождать память так или иначе. Даже если эти функции определяются для использования специализированной системы резервирования памяти, может иметь смысл (для проверки) иметь способность резервировать память тем же способом, что и обычная реализация.

В этом случае можно использовать функции malloc() и free(), унаследованные языком C++ от языка С. Они определяются в заголовке `cstdlib`.

Функция malloc() получает параметр типа `size_t`, задающий количество резервируемых байтов. Она возвращает указатель на зарезервированную область памяти или значение 0, если зарезервировать память не удалось. Функция free() получает параметр типа `void*`, являющийся копией указателя, возвращенного функцией malloc(), и возвращает занятую память операционной системе. Вызов free(0) не делает ничего.

Вот простейший код функций operator new() и operator delete() :

```
void *operator new(size_t size) {
    if (void *mem = malloc(size))
        return mem;
    else
        throw bad_alloc();
}
void operator delete(void *mem) noexcept {
    free(mem);
}
```

Для других версий функции `operator new()` и `operator delete()` код аналогичен.

Упражнения раздела 19.1.1

Упражнение 19.1. Напишите собственную версию функции `operator new(size_t)`, используя функцию `malloc()`, и версию функции `operator delete(void*)`, используя функцию `free()`.

Упражнение 19.2. По умолчанию класс `allocator` использует функцию `operator new()` для резервирования места и функцию `operator delete()` для ее освобождения. Перекомпилируйте и повторно запустите программу `StrVec` (см. раздел 13.5), используя собственные версии функций из предыдущего упражнения.

19.1.2. Размещающий оператор `new`

Хотя функции `operator new()` и `operator delete()` предназначены для использования выражениями `new`, они являются обычными библиотечными функциями. Поэтому обычный код вполне может вызвать их непосредственно.

В прежних версиях языка (до того, как класс `allocator` (см. раздел 12.2.2) стал частью библиотеки), когда необходимо было отделить резервирование от инициализации, использовались функции `operator new()` и `operator delete()`. Эти функции ведут себя аналогично функциям-членам `allocate()` и `deallocate()` класса `allocator` — резервируют и освобождают память, но не создают и не удаляют объекты.

В отличие от класса `allocator`, нет функции `construct()`, позволяющей создавать объекты в памяти, зарезервированной функцией `operator new()`. Вместо этого для создания объекта используется *размещающий оператор new (placement new)* (см. раздел 12.1.2). Как уже упоминалось, эта форма оператора `new` предоставляет дополнительную информацию функции резервирования. Размещающий оператор `new` можно использовать для передачи адреса области. Тогда выражения размещающего оператора `new` будут иметь следующую форму:

```
new (адрес_области) тип
new (адрес_области) тип (инициализаторы)
new (адрес_области) тип [размер]
new (адрес_области) тип [размер] { список
инициализации }
```

где `адрес_области` является указателем, а `инициализаторы` представляют собой разделяемый запятыми список инициализаторов (возможно, пустой), используемый для создания вновь зарезервированного объекта.

Будучи вызванным с адресом, но без других аргументов, размещающий оператор `new` использует вызов `operator new(size_t, void*)` для "резервирования" памяти. Эта версия функции `operator new()` не допускает переопределения (см. раздел 19.1.1). Она *не резервирует* память, а просто возвращает свой аргумент указателя. Затем обычное выражение `new` заканчивает свою работу инициализацией объекта по данному адресу. В действительности размещающий оператор `new` позволяет создать объект в заданной адресом предварительно зарезервированной области памяти.



При передаче одного аргумента, являющегося указателем, выражение размещающего оператора `new` создает объект, но не резервирует память.

Хотя существует несколько способов использования размещающего оператора `new`, он похож на функцию-член `construct()` класса `allocator`, но с одним важным отличием. Передаваемый функции `construct()` указатель должен указывать на область, зарезервированную тем же объектом класса `allocator`. Указатель, передаваемый размещающему оператору `new`, не обязан указывать на область памяти, зарезервированной функцией `operator new()`. Как будет продемонстрировано в разделе 19.6, переданный выражению размещающего оператора `new` указатель даже не обязан указывать на динамическую память.

Явный вызов деструктора

Подобно тому, как размещающий оператор `new` является низкоуровневой альтернативой функции-члену `allocate()` класса `allocator`, явный вызов деструктора аналогичен вызову функции `destroy()`.

Вызов деструктора происходит таким же образом, как и любой другой функции-члена объекта: через указатель или ссылку на объект:

```
string *sp = new string("a value"); // резервирует
```

и инициализирует

// строку

`sp->~string();`

Здесь деструктор вызывается непосредственно. Для получения объекта, на который указывает указатель `sp`, используется оператор стрелки. Затем происходит вызов деструктора, имя которого совпадает с именем типа, но с предваряющим знаком тильды (~).

Подобно вызову функции `destroy()`, вызов деструктора освобождает заданный объект, но не освобождает область, в которой располагается этот объект. При желании эту область можно использовать многократно.



Вызов деструктора удаляет объект, но не освобождает память.

19.2. Идентификация типов времени выполнения

Идентификацию типов времени выполнения (run-time type identification RTTI) обеспечивают два оператора.

- Оператор `typeid`, возвращающий фактический тип заданного выражения.
- Оператор `dynamic_cast`, безопасно преобразующий указатель или ссылку на базовый тип в указатель или ссылку на производный.

Будучи примененными к указателям или ссылкам на тип с виртуальными функциями, эти операторы используют динамический тип (см. раздел 15.2.3) объекта, с которым связан указатель или ссылка.

Эти операторы полезны в случае, когда в производном классе имеется функция, которую необходимо выполнить через указатель или ссылку на объект базового класса, и эту функцию невозможно сделать виртуальной. Обычно по возможности лучше использовать виртуальные функции. Когда применяется виртуальная функция, компилятор автоматически выбирает правильную функцию согласно динамическому типу объекта.

Но определить виртуальную функцию не всегда возможно. В таком случае может пригодиться один из операторов RTTI. С другой стороны, эти операторы более склонны к ошибкам, чем виртуальные функции-члены: разработчик должен знать, к какому типу следует привести объект, и обеспечить проверку успешности приведения.



ВНИМАНИЕ

Динамическое приведение следует использовать осторожно. При каждой возможности желательно создавать и использовать виртуальные функции, а не прибегать к непосредственному управлению типами.

19.2.1. Оператор `dynamic_cast`

Оператор `dynamic_cast` имеет следующую форму:

```
dynamic_cast<тип*>( e )
dynamic_cast<тип&>( e )
dynamic_cast<тип&&>( e )
```

где *тип* должен быть типом класса, у которого (обычно) есть виртуальные функции. В первом случае *e* — допустимый указатель (см. раздел 2.3.2); во втором — l-значение, а в третьем — не должен быть l-

значением.

Во всех случаях тип указателя *e* должен быть либо типом класса, открыто унаследованным от *типа назначения*, либо открытым базовым классом *типа назначения*, либо самим *типовом назначения*. Если указатель *e* будет одним из этих типов, то приведение окажется успешным. В противном случае приведение закончится ошибкой. При неудаче приведения к типу указателя оператор *dynamic_cast* возвращает 0. При неудаче приведения к типу ссылки он передает исключение типа *bad_cast*.

Приведение dynamic_cast для типа указателя

Для примера рассмотрим класс *Base*, обладающий по крайней мере одной виртуальной функцией-членом, и класс *Derived*, открыто унаследованный от класса *Base*. Если имеется указатель *bp* на класс *Base*, то во время выполнения можно привести его к указателю на тип *Derived* следующим образом:

```
if (Derived *dp = dynamic_cast<Derived*>( bp ) ) {  
    // использование объекта Derived, на который  
указывает dp  
} else { // bp указывает на объект Base  
    // использование объекта Base, на который  
указывает dp  
}
```

Если *bp* указывает на объект класса *Derived*, то приведение инициализирует указатель *dp* так, чтобы он указывал на объект класса *Derived*, на который указывает указатель *bp*. В данном случае для кода в операторе *if* вполне безопасно использовать функции класса *Derived*. В противном случае результатом приведения будет 0. Если указатель *dp* нулевой, условие оператора *if* не выполняется. В этом случае блок директивы *else* осуществляет действия, соответствующие классу *Base*.



Оператор *dynamic_cast* применим и к нулевому указателю; результат — пустой указатель требуемого типа.

Обратите внимание на то, что указатель *dp* определен в условии. При

определении переменной в условии приведение и соответствующая проверка осуществляются как единая операция. Кроме того, указатель `dr` недоступен вне оператора `if`. Если приведение потерпит неудачу, то несвязанный указатель не будет доступен для использования в последующем коде, где уже будет забыто успешно ли приведение или нет.

Рекомендуем

Выполнение оператора `dynamic_cast` в условии гарантирует, что приведение и проверка его результата будут осуществлены в одном выражении.

Приведение `dynamic_cast` для типа ссылки

Приведение `dynamic_cast` для ссылочного типа отличается от такового для типа указателя способом сообщения об ошибке. Поскольку нет такого понятия, как пустая ссылка, для них невозможно использовать ту же стратегию сообщений об ошибке, что и для указателей. Когда приведение к ссылочному типу терпит неудачу, передается исключение `std::bad_cast`, определенное в библиотечном заголовке `typeinfo`.

Предыдущий пример можно переписать так, чтобы использовать ссылки следующим образом:

```
void f( const Base &b) {  
    try {  
        const Derived &d = dynamic_cast<const Derived&>  
( b);  
        // использование объекта Derived, на который  
// ссылается b  
    } catch (bad_cast) {  
        // обработка события неудачи приведения  
    }  
}
```

Упражнения раздела 19.2.1

Упражнение 19.3. С учетом следующей иерархии классов, где каждый класс определяет открытый стандартный конструктор и виртуальный деструктор:

```
class A {/*...*/};  
class B : public A { /* ... */ };
```

```
class C : public B { /* ... */ };
class D : public B, public A { /* ... */ };
укажите ошибочные операторы dynamic_cast (если таковые имеются).
```

- (a) A *pa = new C;
 B *pb = dynamic_cast<B*>(pa);
- (b) B *pb = new B;
 C *pc = dynamic_cast<C*>(pb);
- (c) A *pa = new D;
 B *pb = dynamic_cast<B*>(pa);

Упражнение 19.4. Используя классы, определенные в первом упражнении, перепишите следующий код так, чтобы преобразовать выражение *pa в тип C&:

```
if ( C *pc = dynamic_cast<C*>( pa) )
    // используются члены класса C
} else {
    // используются члены класса A
}
```

Упражнение 19.5. Когда стоит использовать оператор `dynamic_cast` вместо виртуальной функции?

19.2.2. Оператор `typeid`

Второй оператор поддержки RTTI — это оператор `typeid`. Оператор `typeid` позволяет выяснить текущий тип объекта.

Выражение `typeid` имеет форму `typeid(e)`, где `e` — любое выражение или имя типа. Результатом оператора `typeid` является ссылка на константный объект библиотечного типа `type_info` или типа, открыто производного от него. В разделе 19.2.4 этот тип рассматривается более подробно. Класс `type_info` определен в заголовке `typeinfo.h`.

Оператор `typeid` применим к выражениям любого типа. Как обычно, спецификатор `const` верхнего уровня (см. раздел 2.4.3) игнорируется, и если выражение является ссылкой, то оператор `typeid` возвращает тип, на который ссылается ссылка. Но при применении к массиву или функции стандартное преобразование в указатель (см. раздел 4.11.2) не осуществляется. Таким образом, результат выражения `typeid(a)`, где `a` является массивом, описывает тип массива, а не тип указателя.

Когда operand не имеет типа класса или является классом без

виртуальных функций, оператор typeid возвращает статический тип операнда. Когда operand является l-значением типа класса, определяющим по крайней мере одну виртуальную функцию, тип результата вычисляется во время выполнения.

Использование оператора typeid

Чаще всего оператор typeid используют для сравнения типов двух выражений или для сравнения типа выражения с определенным типом:

```
Derived *dp = new Derived;  
Base *bp = dp; // оба указателя указывают на объект  
Derived  
// сравнить типы двух объектов во время выполнения  
if (typeid(*bp) == typeid(*dp)) {  
    // bp и dp указывают на объекты того же типа  
}  
// проверить, совпадает ли тип времени выполнения с  
указанным типом  
if (typeid(*bp) == typeid(Derived)) {  
    // bp на самом деле указывает на класс Derived  
}
```

В первом операторе if сравниваются динамические типы объектов, на которые указывают указатели bp и dp. Если оба указателя указывают на тот же тип, то условие истинно. Точно так же второй оператор if истин, если указатель bp в настоящее время указывает на объект класса Derived.

Обратите внимание: operandами оператора typeid являются проверяемые объекты (*bp), а не указатели (bp).

```
// результат проверки всегда ложный: тип bp -  
указатель на класс Base  
if (typeid(bp) == typeid(Derived)) {  
    // код, который никогда не будет выполнен  
}
```

Это условие сравнивает тип Base* с типом Derived. Хотя указатель указывает на объект типа класса, обладающего виртуальными функциями, сам указатель не является объектом типа класса. Тип Base* может быть вычислен и вычисляется во время компиляции. Этот тип не совпадает с типом Derived, поэтому условие всегда будет ложно, независимо от типа объекта, на который указывает указатель bp.



Применение оператора `typeid` к указателю (в отличие от объекта, на который указывает указатель) возвращает статический тип времени компиляции указателя.

Оператор `typeid` требует, чтобы проверка во время выполнения определила, обрабатывается ли выражение. Компилятор обрабатывает выражение, только если у типа есть виртуальные функции. Если у типа нет никаких виртуальных функций, то оператор `typeid` возвращает статический тип выражения; статический тип известен компилятору и без вычисления выражения.

Если динамический тип выражения может отличаться от статического, то выражение следует вычислить (во время выполнения), чтобы определить результирующий тип. Это различие имеет значение при выполнении оператора `typeid(*p)`. Если `p` указывает на тип без виртуальных функций, то указатель `p` не обязан быть допустимым указателем. В противном случае выражение `*p` вычисляется во время выполнения, тогда указатель `p` обязан быть допустимым. Если указатель `p` пуст, то выражение `typeid(*p)` передаст исключение `bad_typeid`.

Упражнения раздела 19.2.2

Упражнение 19.6. Напишите выражение для динамического приведения указателя на тип `Query_base` к указателю на тип `AndQuery` (см. раздел 15.9.1). Проверьте приведение, используя объект класса `AndQuery` и класса другого запроса. Выведите сообщение, подтверждающее работоспособность приведения, и убедитесь, что вывод соответствует ожиданиям.

Упражнение 19.7. Напишите то же приведение, но приведите объект класса `Query_base` к ссылке на тип `AndQuery`. Повторите проверку и удостоверьтесь в правильности работы приведения.

Упражнение 19.8. Напишите выражение `typeid`, чтобы убедиться, указывают ли два указателя на класс `Query_base` на тот же тип. Затем проверьте, не является ли этот тип классом `AndQuery`.

19.2.3. Использование RTTI

В качестве примера случая, когда может пригодиться RTTI, рассмотрим иерархию класса, для которого желательно реализовать оператор равенства (см. раздел 14.3.1). Два объекта равны, если у них тот же тип и то же значение для заданного набора переменных-членов. Каждый производный тип может добавлять собственные данные, которые придется включать в набор проверяемых на равенство.

Казалось бы, эту проблему можно решить, определив набор виртуальных функций, которые проверяют равенство на каждом уровне иерархии. Сделав оператор равенства виртуальным, можно было бы определить одну функцию, которая работает со ссылкой на базовый класс. Этот оператор мог бы передать свою работу виртуальной функции `equal()`, которая и осуществляла бы все необходимые действия.

К сожалению, виртуальные функции не очень хороши для решения этой задачи. Параметры виртуальной функции должны иметь одинаковые типы и в базовом, и в производных классах (см. раздел 15.3). Если бы пришлось определить виртуальную функцию `equal()`, то ее параметр был бы ссылкой на базовый класс. Если параметр является ссылкой на базовый класс, то функция `equal()` сможет использовать только члены из базового класса. Функция `equal()` никак не могла бы сравнить члены, определенные в производном классе.

Оператор равенства должен возвращать значение `false` при попытке сравнить объекты разных типов. Например, если попытаться сравнивать объект базового класса с объектом производного, оператор `==` должен возвратить значение `false`.

С учетом этого наблюдения можно прийти к выводу, что решить данную проблему можно с использованием RTTI. Определим оператор равенства, параметр которого будет ссылкой на тип базового класса. Оператор равенства будет использовать оператор `typeid` для проверки наличия у operandов одинакового типа. Если тип operandов разный, оператор возвратит значение `false`. В противном случае он возвратит виртуальную функцию `equal()`. Каждый класс определит функцию `equal()` так, чтобы сравнить переменные-члены собственного типа. Эти операторы получают параметр типа `Base&`, но приводят operand к собственному типу, прежде чем начать сравнение.

Иерархия класса

Чтобы сделать концепцию более конкретной, предположим, что рассматриваемые классы выглядят следующим образом:

```

class Base {
    friend bool operator==(const Base&, const Base&);
public:
    // члены интерфейса для класса Base
protected:
    virtual bool equal(const Base&) const;
    // данные и другие члены реализации класса Base
};

class Derived: public Base {
public:
    // данные и другие члены реализации класса Base
protected:
    bool equal(const Base&) const;
    // данные и другие члены реализации класса Derived
};

```

Оператор равенства, чувствительный к типу

Рассмотрим, как можно было бы определить общий оператор равенства:

```

bool operator==(const Base &lhs, const Base &rhs) {
    // возвращает false, если типы не совпадают; в
противном случае вызов
    // виртуальной функции equal()
    return typeid(lhs) == typeid(rhs) &&
lhs.equal(rhs);
}

```

Этот оператор возвращает значение `false`, если операнды имеют разный тип. Если они имеют одинаковый тип, оператор делегирует реальную работу по сравнению operandов виртуальной функции `equal()`. Если operandы являются объектами класса `Base`, вызывается функция `Base::equal()`, а если объектами класса `Derived` — то функция `Derived::equal()`.

Виртуальная функция equal()

Каждый класс иерархии должен иметь собственную версию функции `equal()`. Начало у функций всех производных классов будет одинаковым: они приводят аргумент к типу собственного класса:

```

bool Derived::equal(const Base &rhs) const {
    // известно, что типы равны, значит, приведение не

```

```

передаст
    // исключения
    auto r = dynamic_cast<const Derived&>(rhs);
    // действия по сравнению двух объектов класса
Derived и возвращению
    // результата
}

```

Приведение всегда должно быть успешным, ведь оператор равенства вызывает эти функции только после проверки того, что два операнда имеют одинаковый тип. Однако приведение необходимо, чтобы функция могла обращаться к производным членам правого операнда.

Функция `equal()` базового класса

Эта функция гораздо проще других:

```

bool Base::equal(const Base &rhs) const {
    // действия по сравнению двух объектов класса Base
}

```

Здесь нет никакой необходимости в приведении аргументов перед применением. Оба они, и `*this` и параметр, являются объектами класса `Base`, поэтому все доступные для него функции содержатся в классе объекта.

19.2.4. Класс `type_info`

Точное определение класса `type_info` зависит от компилятора, но стандарт гарантирует, что класс будет определен в заголовке `typeinfo` и предоставлять, по крайней мере, те функции, которые перечислены в табл. 19.1.

Этот класс обладает также открытым виртуальным деструктором, поскольку он предназначен для использования в качестве базового класса. Если компилятор позволяет предоставить дополнительную информацию о типе, для этого следует воспользоваться классом, производным от класса `type_info`.

Таблица 19.1. Функции класса `type_info`

<code>t1 == t2</code>	Возвращает значение <code>true</code> , если оба объекта (<code>t1</code> и <code>t2</code>) имеют тот же тип, и значение <code>false</code> — в противном случае
<code>t1 != t2</code>	Возвращает значение <code>true</code> , если оба объекта (<code>t1</code> и <code>t2</code>) имеют разные типы, и значение <code>false</code> — в противном случае

t. name()	Возвращает символьную строку в стиле С, содержащую отображаемую версию имени типа. Имена типов создаются способом, не зависящим от системы
t1. before(t2)	Возвращает логическое значение (тип <code>bool</code>), указывающее на то, следует ли тип <code>t1</code> прежде типа <code>t2</code> . Порядок следования зависит от компилятора

У класса `type_info` нет стандартного конструктора, а оператор присвоения, конструктор копий и перемещения определены как удаленные (см. раздел 13.1.6). Поэтому нельзя определять, копировать или присваивать объекты типа `type_info`. Единственный способ создания объектов класса `type_info` — это оператор `typeid`.

Функция-член `name()` возвращает символьную строку в стиле С, содержащую имя класса объекта. Значение, используемое для данного типа, зависит от компилятора и не обязательно соответствует имени класса, использованному в программе. Единственное, что гарантирует функция `name()`, — это уникальность возвращаемой ей строки для данного типа.

Рассмотрим пример:

```
int arr[10];
Derived d;
Base *p = &d;
cout << typeid(42).name() << ", "
    << typeid(arr).name() << ", "
    << typeid(Sales_data).name() << ", "
    << typeid(std::string).name() << ", "
    << typeid(p).name() << ", "
    << typeid(*p).name() << endl;
```

При запуске на машине авторов эта программа выводит следующее
`i, A10_i, 10Sales_data, Ss, P4Base, 7Derived`



Класс `type_info` зависит от компилятора. Некоторые компиляторы предоставляют и другие функции-члены, которые возвращают дополнительную информацию о типах, используемых в программе. Чтобы выяснить реальные возможности класса `type_info` для конкретного компилятора, необходимо обратиться к его документации.

Упражнения раздела 19.2.4

Упражнение 19.9. Напишите программу, подобную приведенной в конце этого раздела, для вывода имен, используемых компилятором для общих типов. Если ваш компилятор создает вывод, подобный нашему, напишите функцию, которая преобразует эти строки в более понятную для человека форму.

Упражнение 19.10. С учетом приведенной ниже иерархии классов, в которой каждый класс обладает открытым стандартным конструктором и виртуальным деструктором, укажите, какие имена типов отобразят следующие операторы?

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public B { /* ... */ };

(a) A *pa = new C;
    cout << typeid( pa ).name() << endl;
(b) C cobj;
    A& ra = cobj;
    cout << typeid( &ra ).name() << endl;
(c) B *px = new B;
    A& ra = *px;
    cout << typeid( ra ).name() << endl;
```

19.3. Перечисления

Перечисления (enumeration) позволяют группировать наборы целочисленных констант. Как и класс, каждое перечисление определяет новый тип. Перечисления — литеральные типы (см. раздел 7.5.6).



В языке C++ есть два вида перечислений: с ограниченной и с не ограниченной областью видимости. *Перечисление с ограниченной областью видимости* (scoped enumeration) вводит новый стандарт. Для определения перечисления с ограниченной областью видимости используются ключевые слова `enum class` (или `enum struct`), сопровождаемые именем перечисления и разделяемым запятыми списком перечислителей (enumerator), заключенным в фигурные скобки. За закрывающей фигурной скобкой следует точка с запятой:

```
enum class open_modes { input, output, append};
```

Здесь определен тип перечисления `open_modes` с тремя перечислителями: `input`, `output` и `append`.

В определении *перечисления с не ограниченной областью видимости* (unscoped enumeration) ключевое слово `class` (или `struct`) отсутствует. Имя перечисления с не ограниченной областью видимости не является обязательным:

```
enum color { red, yellow, green}; // перечисление с  
не ограниченной
```

// областью

видимости

```
// безымянное перечисление с не ограниченной  
областью видимости
```

```
enum { floatPrec = 6, doublePrec = 10,  
double_doublePrec = 10};
```

Если перечисление является безымянным, определить объекты его типа можно только в составе определения перечисления. Подобно определению класса, здесь можно предоставить разделяемый запятыми список объявлений между закрывающей фигурной скобкой и точкой с запятой, завершающей определение перечисления (см. раздел 2.6.1).

Перечислители

Имена перечислителей в перечислении с ограниченной областью видимости подчиняются обычным правилам областей видимости и недоступны вне области видимости перечисления. Имена перечислителей в перечислении с не ограниченной областью видимости находятся в той же области видимости, что и само перечисление:

```
enum color { red, yellow, green}; // перечисление с
не ограниченной // областью
видимости
enum stoplight { red, yellow, green}; // ошибка:
переопределение // перечислителей
enum class peppers { red, yellow, green}; // ok:
перечислители
// скрываются
color eyes = green; // ok: перечислители находятся
в области видимости // для перечисления с не ограниченной
областью видимости
peppers p = green; // ошибка: перечислители из
peppers не находятся в // области видимости
// color::green находится в
области видимости,
// но имеет неправильный тип
color hair = color::red; // ok: к перечислителям
можно обратиться явно
peppers p2 = peppers::red; // ok: использование red
из peppers
```

По умолчанию значения перечислителей начинаются с 0, и значение каждого последующего перечислителя на 1 больше предыдущего. Однако вполне можно предоставить инициализаторы для одного или нескольких перечислителей:

```
enum class intTypes {
    charTyp = 8, shortTyp = 16, intTyp = 16,
    longTyp = 32, long_longTyp = 64
};
```

Как можно заметить на примере перечислителей `intTyp` и `shortTyp`, значение перечислителя не обязано быть уникальным. Без инициализатора значение перечислителя будет на 1 больше, чем у предыдущего.

Перечислители являются константами, и их инициализаторы должны быть константными выражениями (см. раздел 2.4.4). Следовательно, каждый перечислитель сам является константным выражением. Поскольку перечислители — константные выражения, их можно использовать там, где необходимы константные выражения. Например, можно определить переменные `constexpr` типа перечисления:

```
constexpr intTypes charbits = intTypes::charTyp;
```

Точно так же перечисление можно использовать как выражение в операторе `switch`, а значения его перечислителей как метки разделов `case` (см. раздел 5.3.2). По той же причине тип перечисления можно также использовать как параметр значения шаблона (см. раздел 16.1.1) и инициализировать статические переменные-члены типа перечисления в определении класса (см. раздел 7.6).

Подобно классам, перечисления определяют новые типы

Поскольку перечисление имеет имя, можно определять и инициализировать объекты этого типа. Объект перечисления может быть инициализирован или присвоен только одному из своих перечислителей или другому объекту того же типа перечисления:

```
open_modes om = 2;           // ошибка: 2 не имеет типа
open_modes
om      = open_modes::input;   // ok: input - 
перечислитель open_modes
```

Объекты или перечислители типа перечисления с не ограниченной областью видимости автоматически преобразовываются в целочисленный тип. В результате они применимы там, где требуется целочисленное значение:

```
int i = color::red;          // ok: перечислитель
перечисления с не ограниченной
// областью видимости неявно
преобразован в тип int
int j = peppers::red;        // ошибка: перечисления с
ограниченной областью
// видимости неявно не
преобразуются
```

Определение размера перечисления



Хотя каждое перечисление определяет уникальный тип, оно представляется одним из встроенных целочисленных типов. По новому стандарту можно указать, что следует использовать тип, заданный за именем перечисления и двоеточием:

```
enum intValues : unsigned long long {  
    charTyp = 255, shortTyp = 65535, intTyp = 65535,  
    longTyp = 4294967295UL,  
    long_longTyp = 18446744073709551615ULL  
};
```

Если базовый тип не задан, то по умолчанию перечисления с ограниченной областью видимости имеют базовый тип `int`. Для перечислений с не ограниченной областью видимости типа по умолчанию нет; известно только то, что базовый тип достаточно велик для содержания значения перечислителя. Когда базовый тип определяется (включая неявное определение для перечисления с ограниченной областью видимости), попытка создания перечислителя, значение которого превосходит заданный тип, приведет к ошибке.

Возможность определить базовый тип перечисления позволяет контролировать тип, используемый при разных реализациях компилятора. Это позволяет также гарантировать, что программа, откомпилированная на одной реализации, создаст тот же код при компиляции на другом.

Предварительные объявления для перечислений



По новому стандарту перечисление можно объявить предварительно. Предварительное объявление перечисления должно определить (неявно или явно) его базовый размер:

```
// предварительное объявление перечисления с не  
ограниченной областью  
// видимости intValues  
enum intValues : unsigned long long; //  
перечисление с не ограниченной // областью видимости
```

```
должно определять тип
enum class open_modes;      // перечисление с
ограниченной областью
                           // видимости может использовать по
умолчанию тип int
```

Поскольку для перечисления с не ограниченной областью видимости нет размера по умолчанию, каждое объявление должно включить его размер. Перечисление с ограниченной областью видимости можно объявить, не определяя размер, тогда размер неявно определяется как `int`.

Подобно любым объявлениям, все объявления и определения того же перечисления должны соответствовать друг другу. В случае перечислений это требование означает, что размер перечисления должен совпадать для всех объявлений и определений. Кроме того, нельзя объявить имя как перечисление с не ограниченной областью видимости в одном контексте, а затем повторно объявить его как перечисление с ограниченной областью видимости:

```
// ошибка: в объявлении и определении должно
совпадать, ограничена ли
// область видимости перечисления
enum class intValues;
enum intValues; // ошибка: intValues ранее
объявлено как перечисление с
                           // ограниченной областью видимости
enum intValues : long; // ошибка: intValues ранее
объявлено как int
```

Соответствие параметров и перечисления

Поскольку объект типа перечисления может быть инициализирован только другим объектом того же типа перечисления или одним из его перечислителей (см. раздел 19.3), целое число, значение которого случайно совпадает со значением перечислителя, не может использоваться при вызове функции, ожидающей перечислимый аргумент:

```
// перечисление с не ограниченной областью
видимости;
// базовый тип зависит от машины
enum Tokens { INLINE = 128, VIRTUAL = 129 };
void ff(Tokens);
void ff(int);
int main() {
```

```
Tokens curTok = INLINE;
ff(128); // точно соответствует ff( int )
ff(INLINE); // точно соответствует ff( Tokens )
ff(curTok); // точно соответствует ff( Tokens )
return 0;
}
```

Хоть и нельзя передать целочисленное значение параметру перечислимого типа, вполне можно передать объект или перечислитель перечисления с неограниченной областью видимости параметру целочисленного типа. При этом значение перечислителя преобразуется в тип `int` или больший целочисленный тип. Фактический тип преобразования зависит от базового типа перечисления:

```
void newf( unsigned char );
void newf( int );
unsigned char uc = VIRTUAL;
newf( VIRTUAL ); // вызов newf( int )
newf( uc ); // вызов newf( unsigned char )
```

У перечисления `Tokens` только два перечислителя, больший из них имеет значение 129. Это значение может быть представлено типом `unsigned char`, и большинство компиляторов будут использовать для перечисления `Tokens` базовый тип `unsigned char`. Независимо от своего базового типа, объекты и перечислители перечисления `Tokens` преобразуются в тип `int`. Перечислители и значения перечислимого типа не преобразуются в тип `unsigned char`, даже если ему соответствуют значения перечислителей.

19.4. Указатель на член класса

Указатель на член класса (pointer to member) — это указатель, способный указывать на нестатический член класса. Обычно указатель указывает на объект, но указатель на член класса идентифицирует только член класса объекта, а не весь объект. Статические члены класса не являются частью конкретного объекта, поэтому для указания на них не нужен никакой специальный синтаксис. Указатели на статические члены являются обычными указателями.

Тип указателя на член класса объединяет тип класса и тип члена этого класса. Такие указатели инициализируют как указывающие на определенный член класса, не указывая объект, которому принадлежит этот член. При применении указателя на член класса предоставляется объект, член класса которого предстоит использовать.

Для демонстрации работы указателей на члены класса воспользуемся упрощенной версией класса Screen из раздела 7.3.1:

```
class Screen {  
public:  
    typedef std::string::size_type pos;  
    char get_cursor() const { return contents[cursor];}  
    char get() const;  
    char get( pos ht, pos wd) const;  
private:  
    std::string contents;  
    pos cursor;  
    pos height, width;  
};
```

19.4.1. Указатели на переменные-члены

Подобно любым указателям, при объявлении указателя на член класса используется символ *, означающий, что объявляемое имя является указателем. В отличие от обычных указателей, указатель на член класса включает также имя класса, содержащего этот член. Следовательно, символу * должна предшествовать часть *имяКласса*::, означающая, что определяемый указатель способен указывать на член класса *имяКласса*. Например:

```
// pdata может указывать на член типа string
контантного (или не
// константного) объекта класса Screen
const string Screen:: *pdata;
```

Приведенный выше код объявляет pdata "указателем на член класса Screen, обладающий типом const string". Переменные-члены константного объекта сами являются константами. Объявление указателя pdata как указателя на тип const string позволяет использовать его для указания на член любого объекта класса Screen, константного или нет. Взамен указатель pdata применим только для чтения, но не для записи в член класса, на который он указывает.

При инициализации (или присвоении) указателя на член класса следует заявить, на который член он указывает. Например, можно заставить указать pdata указывать на переменную-член contents неопределенного объекта класса Screen следующим образом:

```
pdata = &Screen:: contents;
```

Здесь оператор обращения к адресу применяется не к объекту в памяти, а к члену класса Screen.

Конечно, по новому стандарту проще объявить указатель на член класса при помощи ключевых слов auto или decltype:

```
auto pdata = &Screen:: contents;
```

Использование указателей на переменные-члены

Важно понять, что при инициализации или присвоении указателя на член класса он еще не указывает на данные. Он идентифицирует определенный член класса, но не содержащий его объект. Объект предоставляется при обращении к значению указателя на член класса.

Подобно *операторам доступа к членам* (member access operator), . и ->, существуют два оператора доступа к указателю на член класса, . * и ->*, позволяющие предоставить объект и обращаться к значению указателя для доступа к члену этого объекта:

```
Screen myScreen, *pScreen = &myScreen;
// .* обращение к значению pdata для доступа к
содержимому члена данного
// объекта класса myScreen
auto s = myScreen. *pdata;
// ->* обращение к значению pdata для доступа к
содержимому члена
```

```
// объекта, на который указывает pScreen  
s = pScreen->*pdata;
```

Концептуально эти операторы выполняют два действия: обращаются к значению указателя на член класса, чтобы получить доступ к необходимому члену; затем, подобно операторам обращения к членам, они обращаются к члену данного объекта непосредственно (. *) или через указатель (->*).

Функция, возвращающая указатель на переменную-член

К указателям на члены применимы обычные средства управления доступом. Например, член `contents` класса `Screen` является закрытым. В результате указатель `pdata` выше должен использоваться в члене класса `Screen`, его дружественном классе, либо произойдет ошибка.

Поскольку переменные-члены обычно являются закрытыми, как правило, нельзя получать указатель на саму переменную-член. Вместо этого, если такой класс, как `Screen`, желает предоставить доступ к своему члену `contents`, то он определил бы функцию, возвращающую указатель на эту переменную-член:

```
class Screen {  
public:  
    // data() - статический член, возвращающий  
    // указатель на член класса  
    static const std::string Screen::*data()  
    { return &Screen::contents; }  
    // другие члены, как прежде  
};
```

Здесь в класс `Screen` добавлена статическая функция-член, возвращающая указатель на переменную-член `contents` класса `Screen`. Тип возвращаемого значения этой функции совпадает с типом первоначального указателя `pdata`. Читая тип возвращаемого значения справа налево, можно заметить, что функция `data()` возвращает указатель на член класса `Screen`, имеющий тип `string` и являющийся константой. Тело функции применяет оператор обращения к адресу к переменной-члену `contents`. Таким образом, функция возвращает указатель на переменную-член `contents` класса `Screen`.

Когда происходит вызов функции `data()`, возвращается указатель на член класса:

```
// data() возвращает указатель на член contents
```

класса Screen

```
const string Screen::*pdata = Screen::data();
```

Как и прежде, указатель pdata указывает на член класса Screen, но не на фактические данные. Чтобы использовать указатель pdata, следует связать его с объектом типа Screen:

```
// получить содержимое объекта myScreen
auto s = myScreen.*pdata;
```

Упражнения раздела 19.4.1

Упражнение 19.11. В чем разница между обычным указателем на данные и указателем на переменную-член?

Упражнение 19.12. Определите указатель на член класса, способный указывать на член cursor класса Screen. Получите через этот указатель значение Screen::cursor.

Упражнение 19.13. Определите тип, способный представить указатель на член bookNo класса Sales_data.

19.4.2. Указатели на функции-члены

Вполне можно также определить указатель, способный указывать на функцию-член класса. Подобно указателям на переменные-члены, самый простой способ создания указателя на функцию-член — это использовать ключевое слово `auto` для автоматического выводения типа:

```
// указатель pmf способен указывать на функцию-член
// класса Screen,
// возвращающую тип char и не получающую никаких
// аргументов
auto pmf = &Screen::get_cursor;
```

Как и указатель на переменную-член, указатель на функцию-член объявляется с использованием синтаксиса `имяКласса::*`. Подобно любому другому указателю на функцию (см. раздел 6.7), указатель на функцию-член определяет тип возвращаемого значения и список типов параметров функции, на которую может указывать этот указатель. Если функция-член является константной (см. раздел 7.1.2) или ссылочной (см. раздел 13.6.3), следует также добавить квалификатор `const` или квалификатор ссылки.

Подобно обычным указателям на функцию, если функция-член перегружена, следует явно указать, какая именно функция имеется в виду (см. раздел 6.7). Например, указатель на версию функции `get()` с двумя параметрами можно объявить так:

```
char (Screen::*pmf2)(Screen::pos, Screen::pos)
const;
pmf2 = &Screen::get;
```

Круглые скобки вокруг части `Screen::*` в этом объявлении необходимы из-за приоритета. Без круглых скобок компилятор воспримет следующий код как (недопустимое) объявление функции:

```
// ошибка: у функции, не являющейся членом класса
p, не может быть
// спецификатора const
char Screen::*p(Screen::pos, Screen::pos) const;
```

Это объявление пытается определить обычную функцию по имени `p`, которая возвращает указатель на член класса `Screen` типа `char`. Поскольку объявляется обычная функция, за объявлением не может быть спецификатора `const`.

В отличие от обычных указателей на функцию, нет никакого

автоматического преобразования между функцией-членом и указателем на этот член:

```
// pmf указывает на член класса Screen, не
получающий аргументов и
// возвращающий тип char
pmf = &Screen::get; // нужно явно использовать
оператор обращения к
                           // адресу
pmf = Screen::get; // ошибка: нет преобразования в
указатель для
                           // функций-членов
```

Использование указателя на функцию-член

Как и при использовании указателя на переменную-член, для вызова функции-члена через указатель на член класса используются операторы `*` и `->*`:

```
Screen myScreen, *pScreen = &myScreen;
// вызов функции, на которую указывает указатель
pmf объекта,
// на который указывает указатель pScreen
char c1 = (pScreen->*pmf)();
// передает аргументы 0, 0 версии функции get() с
двумя параметрами
// объекта myScreen
char c2 = (myScreen.*pmf2)(0, 0);
```

Вызовы `(myScreen->*pmf)()` и `(pScreen.*pmf2)(0, 0)` требуют круглых скобок, поскольку приоритет оператора вызова выше, чем приоритет оператора указателя на член класса.

Без круглых скобок вызов `myScreen.*pmf()` был бы интерпретирован как `myScreen.*(pmf())`.

Этот код требует вызвать функцию `pmf()` и использовать ее возвращаемое значение как operand оператора указателя на член класса `(.*)`. Но `pmf` — не функция, поэтому данный код ошибочен.



Из-за разницы приоритетов операторов вызова объявления указателей на функции-члены и вызовы через такие указатели должны использовать

круглые скобки: (C: : * p) (parms) и (obj. * p) (args) .

Использование псевдонимов типов для указателей на члены

Псевдонимы типа или `typedef` (см. раздел 2.5.1) существенно облегчают чтение указателей на члены. Например, следующий код определяет псевдоним типа `Action` как альтернативное имя для типа версии функции `get()` с двумя параметрами:

```
// Action - тип, способный указывать на функцию-
член класса Screen,
// возвращающую тип char и получающую два аргумента
типа pos
using Action =
    char (Screen: : * )( Screen: : pos, Screen: : pos) const;
```

`Action` — это другое имя для типа "указатель на константную функцию-член класса `Screen`, получающую два параметра типа `pos` и возвращающую тип `char`". Используя этот псевдоним, можно упростить определение указателя на функцию `get()` следующим образом:

```
Action get = &Screen: : get; // get указывает на член
get() класса Screen
```

Подобно любым другим указателям на функцию, тип указателя на функцию-член можно использовать как тип возвращаемого значения или как тип параметра функции. Подобно любому другому параметру, у параметра указателя на член класса может быть аргумент по умолчанию:

```
// action() получает ссылку на класс Screen и
указатель на его
// функцию-член
```

```
Screen& action( Screen&, Action = &Screen: : get);
```

Функция `action()` получает два параметра, которые являются ссылками на объект класса `Screen`, и указатель на функцию-член класса `Screen`, получающую два параметра типа `pos` и возвращающую тип `char`. Функцию `action()` можно вызвать, передав ей указатель или адрес соответствующей функции-члена класса `Screen`:

```
Screen myScreen;
// эквивалентные вызовы:
action( myScreen);           // использует аргумент по
умолчанию
action( myScreen, get); // использует предварительно
определенную
```

```
// переменную get  
action( myScreen, &Screen::get); // передает адрес  
явно
```



Псевдонимы типа облегчают чтение и написание кода, использующего указатели.

Таблицы указателей на функцию-член

Как правило, перед использованием указатели на функции и указатели на функции-члены хранят в таблице функций (см. раздел 14.8.3). Когда у класса есть несколько членов того же типа, такая таблица применяется для выбора одного из набора этих членов. Предположим, что класс Screen дополнен некоторыми функциями-членами, каждая из которых перемещает курсор в определенном направлении:

```
class Screen {  
public:  
    // другие члены интерфейса и реализации, как  
прежде  
    Screen& home(); // функции перемещения курсора  
    Screen& forward();  
    Screen& back();  
    Screen& up();  
    Screen& down();  
};
```

Каждая из этих новых функций не получает никаких параметров и возвращает ссылку на вызвавший ее объект класса Screen.

Можно определить функцию move(), способную вызвать любую из этих функций и выполнить указанное действие. Для поддержки этой новой функции в класс Screen добавлен статический член, являющийся массивом указателей на функции перемещения курсора:

```
class Screen {  
public:  
    // другие члены интерфейса и реализации, как  
прежде  
    // Action - указатель, который может быть присвоен
```

любой из

```
// функций-членов перемещения курсора
using Action = Screen&( Screen::*)();
// задать направление перемещения;
// перечисления описаны в разделе 19.3
enum Directions { HOME, FORWARD, BACK, UP, DOWN };
Screen& move(Directions);
private:
    static Action Menu[ ]; // таблица функций
};
```

Массив *Menu* содержит указатели на каждую из функций перемещения курсора. Эти функции будут храниться со смещениями, соответствующими перечислителям перечисления *Directions*. Функция *move()* получает перечислитель и вызывает соответствующую функцию:

```
Screen& Screen::move( Directions cm) {
    // запустить элемент по индексу cm для объекта
    this
    return (this->*Menu[ cm])( ); // Menu[ cm] указывает
на функцию-член
}
```

Вызов *move()* обрабатывается следующим образом: выбирается элемент массива *Menu* по индексу *cm*. Этот элемент является указателем на функцию-член класса *Screen*. Происходит вызов функции-члена, на которую указывает этот элемент от имени объекта, на который указывает указатель *this*.

Когда происходит вызов функции *move()*, ему передается перечислитель, указывающий направление перемещения курсора:

```
Screen myScreen;
myScreen. move( Screen::HOME); // вызывает
myScreen. home
myScreen. move( Screen::DOWN); // вызывает
myScreen. down
```

Остается только определить и инициализировать саму таблицу:

```
Screen::Action Screen::Menu[ ] = { &Screen::home,
                                    &Screen::forward,
                                    &Screen::back,
                                    &Screen::up,
                                    &Screen::down,
```

```
};
```

Упражнения раздела 19.4.2

Упражнение 19.14. Корректен ли следующий код? Если да, то что он делает? Если нет, то почему?

```
auto pmf = &Screen::get_cursor; pmf = &Screen::get;
```

Упражнение 19.15. В чем разница между обычным указателем на функцию и указателем на функцию-член?

Упражнение 19.16. Напишите псевдоним типа, являющийся синонимом для указателя, способного указать на переменную-член avgprice класса Sales_data.

Упражнение 19.17. Определите псевдоним типа для каждого отдельного типа функции-члена класса Screen.

19.4.3. Использование функций-членов как вызываемых объектов

Как уже упоминалось, для вызова через указатель на функцию-член, нужно использовать операторы `.*` и `->*` для связи указателя с определенным объектом. В результате, в отличие от обычных указателей на функцию, указатель на функцию-член класса не является *вызываемым объектом*; эти указатели не поддерживают оператор вызова функции (см. раздел 10.3.2).

Поскольку указатель на член класса не является вызываемым объектом, нельзя непосредственно передать указатель на функцию-член алгоритму. Например, если необходимо найти первую пустую строку в векторе строк, вполне очевидный вызов не сработает:

```
auto fp = &string::empty; // fp указывает на функцию empty()
                           // класса string
// ошибка: для вызова через указатель на член класса следует
// использовать оператор .* или ->*
find_if(svec.begin(), svec.end(), fp);
```

Алгоритм `find_if()` ожидает вызываемый объект, но предоставляется указатель на функцию-член `fp`. Этот вызов не будет откомпилирован, поскольку код в алгоритме `find_if()` выполняет примерно такой оператор:

```
// проверяет применимость данного предиката к
```

```

текущему элементу,
// возвращает true
if (fp(*it)) // ошибка: для вызова через указатель
на член класса
    // следует использовать оператор ->*

```

Использование шаблона *Function* для создания вызываемого объекта

Один из способов получения вызываемого объекта из указателя на функцию-член подразумевает использование библиотечного шаблона *function* (см. раздел 14.8.3):

```

function<bool      (const      string&) >      fcn      =
&string::empty;
    find_if(svec.begin(), svec.end(), fcn);

```

Здесь шаблону *function* указано, что *empty()* — это функция, которая может быть вызвана со строкой и возвращает значение типа *bool*. Обычно объект, для которого выполняется функция-член, передается неявному параметру *this*. Когда шаблон *function* используется при создании вызываемого объекта для функции-члена, следует преобразовать код так, чтобы сделать этот неявный параметр явным.

Когда объект шаблона *function* содержит указатель на функцию-член, класс *function* знает, что для вызова следует использовать соответствующий оператор указателя на член класса. Таким образом, можно предположить, что у функции *find_if()* будет код наподобие следующего:

```

// если it является итератором в функции find_if(),
то *it - объект
// в заданном диапазоне
if (fcn(*it)) // fcn - имя вызываемого объекта в
функции find_if()

```

Его и выполнит шаблон класса *function*, используя соответствующий оператор указателя на член класса. Класс *function* преобразует этот вызов в такой код:

```

// если it является итератором в функции find_if(),
то *it - объект
// в заданном диапазоне
if (((*it).*p)()) // p - указатель на функцию-член
в функции fcn

```

При определении объекта шаблона `function` следует указать тип функции, сигнатура которой определяет представляемые вызываемые объекты. Когда вызываемой объект является функцией-членом, первый параметр сигнатуры должен представить (обычно неявный) объект, для которого будет выполнена функция-член. Передаваемая шаблону `function` сигнатура должна определять, будет ли объект передан как указатель или как ссылка.

При определении вызываемого объекта `fn()` было известно, что нужно вызвать функцию `find_if()` для последовательности строковых объектов. Следовательно, от шаблона `function` требовалось создать вызываемый объект, получающий объекты класса `string`. Если бы вектор содержал указатели на тип `string`, от шаблона `function` требовалось бы ожидать указатель:

```
vector<string*> pvec;
function<bool (const string*)> fp = &string::empty;
// fp получает указатель на string и использует
// оператор ->* для вызова
// функции empty()
find_if( pvec.begin(), pvec.end(), fp);
```

Использование шаблона `mem_fn` для создания вызываемого объекта



Чтобы использовать шаблон `function`, следует предоставить сигнатуру вызова члена, который предстоит вызвать. Но можно позволить компилятору вывести тип функции-члена при использовании другого библиотечного средства, шаблона `mem_fn`, определенного, как и шаблон `function`, в заголовке `functional`. Как и шаблон `function`, шаблон `mem_fn` создает вызываемый объект из указателя на член класса. В отличие от шаблона `function`, шаблон `mem_fn` выведет тип вызываемого объекта из типа указателя на член класса:

```
find_if(svec.begin(),
mem_fn(&string::empty)); svec.end(),
```

Здесь шаблон `mem_fn(&string::empty)` создает вызываемый объект, получающий строковый аргумент и возвращающий логическое значение.

Вызываемый объект, созданный шаблоном `mem_fn`, может быть

вызван для объекта или указателя:

```
auto f = mem_fn(&string::empty); // f получает
string или string*
f(*svec.begin()); // ok: передача объекта string; f
использует .* для
                    // вызова empty()
f(&svec[0]);           // ok: передача указателя на
string; f использует .->
                    // для вызова empty()
```

Фактически шаблон `mem_fn` можно считать как будто создающим вызываемый объект с перегруженным оператором вызова функции — один получает тип `string*`, а другой — `string&`.

Использование функции bind() для создания вызываемого объекта

Для создания вызываемого объекта из функции-члена можно также использовать функцию `bind()` (см. раздел 10.3.4):

```
// связать каждую строку из диапазона
// с неявным первым аргументом empty()
auto it = find_if(svec.begin(), svec.end(),
                  bind(&string::empty, _1));
```

Подобно шаблону `function`, при использовании функции `bind()` следует сделать явным обычно неявный параметр функции-члена, представляющий объект, с которым будет работать функция-член. Подобно шаблону `mem_fn`, первый аргумент вызываемого объекта, создаваемого функцией `bind()`, может быть либо указателем, либо ссылкой на тип `string`:

```
auto f = bind(&string::empty, _1);
f(*svec.begin()); // ok: аргумент — строка f,
использует .* для вызова
                    // функции empty()
f(&svec[0]); // ok: аргумент — указатель на строку
f использует .->
                    // для вызова функции empty()
```

Упражнения раздела 19.4.3

Упражнение 19.18. Напишите функцию, использующую алгоритм `count_if()` для подсчета количества пустых строк в заданном векторе.

Упражнение 19.19. Напишите функцию, получающую вектор

`vector<Sales_data>` и находящую первый элемент, средняя цена которого превосходит заданное значение.

19.5. Вложенные классы

Класс, определяемый в другом классе, называется *вложенным классом* (nested class) или *вложенным типом* (nested type). Вложенные классы обычно используются для классов реализации, как, например, класс `QueryResult` из приложения текстового запроса (см. раздел 12.3).

Имя вложенного класса видимо в области видимости содержащего его класса, но не вне ее. Имя вложенного класса не будет входить в конфликт с тем же именем, объявленным в другой области видимости.

Вложенный класс может содержать члены тех же видов, что и не вложенный класс. Подобно любому другому классу, вложенный класс контролирует доступ к своим членам при помощи спецификаторов доступа. Содержащий класс не имеет никаких специальных прав доступа к членам вложенного класса, а вложенный класс не имеет привилегий в доступе к членам содержащего его класса.

В содержащем классе вложенный класс представляет собой член, типом которого является класс. Подобно любому другому члену, содержащий класс задает уровень доступа к этому типу. Вложенный класс, определенный в разделе `public` содержащего класса, может быть использован везде. Вложенный класс, определенный в разделе `protected`, доступен только содержащему классу, его производным и дружественным классам. Вложенный класс, определенный в разделе `private`, доступен лишь для членов содержащего класса и классов, дружественных для него.

Объявление вложенного класса

Класс `TextQuery` из раздела 12.3.2 определял сопутствующий класс `QueryResult`. Класс `QueryResult` жестко связан с классом `TextQuery`. Класс `QueryResult` имел бы смысл использовать и для других целей, а не только для результатов операции запроса к объекту класса `TextQuery`. Для отражения этой жесткой связи сделаем класс `QueryResult` членом класса `TextQuery`.

```
class TextQuery {  
public:  
    class QueryResult; // вложенный класс будет  
определён позже
```

```
// другие члены, как в разделе 12.3.2  
};
```

В первоначальный класс `TextQuery` необходимо внести только одно изменение — объявить о намерении определить класс `QueryResult` как вложенный. Поскольку класс `QueryResult` будет типом-членом (см. раздел 7.3.4), его следует объявить прежде, чем использовать. В частности, класс `QueryResult` следует объявить прежде, чем использовать его как тип возвращаемого значения функции-члена `query()`. Остальные члены первоначального класса неизменны.

Определение вложенного класса вне содержащего класса

В классе `TextQuery` класс `QueryResult` объявлен, но не определен. Подобно функциям-членам, вложенные классы следует объявить в классе, но определен он может быть в или вне класса.

При определении вложенного класса вне его содержащего класса следует квалифицировать имя вложенного класса именем его содержащего класса:

```
// определение класса QueryResult как члена класса  
TextQuery  
class TextQuery::QueryResult {  
    // в области видимости класса не нужно  
квалифицировать имя  
    // параметров QueryResult  
    friend std::ostream&  
        print(std::ostream&, const QueryResult&);  
public:  
    // не нужно определять QueryResult::line_no;  
вложенный класс способен  
    // использовать член своего содержащего класса без  
необходимости  
    // квалифицировать его имя  
    QueryResult(std::string,  
        std::shared_ptr<std::set<line_no>>,  
        std::shared_ptr<std::vector<std::string>>);  
    // другие члены, как в разделе 12.3.2  
};
```

Единственное изменение, внесенное в первоначальный класс, заключается в том, что в классе `QueryResult` больше не определяется

переменная-член `line_no`. Члены класса `QueryResult` могут обращаться к этому имени непосредственно в классе `TextQuery`, таким образом, нет никакой необходимости определять его снова.



Пока не встретится фактическое определение вложенного класса, расположенное вне тела класса, этот класс является незавершенным типом (см. раздел 7.3.3).

Определение членов вложенного класса

В этой версии конструктор `QueryResult()` не определяется в теле класса. Чтобы определить конструктор, следует указать, что класс `QueryResult` вложен в пределы класса `TextQuery`. Для этого имя вложенного класса квалифицируют именем содержащего его класса:

```
// определение члена класса по имени QueryResult
для класса по
// имени QueryResult, вложенного в класс TextQuery
TextQuery::QueryResult::QueryResult(string s,
```

```
shared_ptr<set<line_no>> p,
```

```
shared_ptr<vector<string>> f):
    sought(s), lines(p), file(f) {}
```

Читая имя функции справа налево, можно заметить, что это определение конструктора для класса `QueryResult`, который вложен в пределы класса `TextQuery`. Сам код только сохраняет данные аргументов в переменных-членах и не делает больше ничего.

Определение статических членов вложенных классов

Если бы класс `QueryResult` объявлял статический член, его определение находилось бы вне области видимости класса `TextQuery`. Например, статический член класса `QueryResult` был бы определен как-то так:

```
// определение статического члена типа int класса
QueryResult
// вложенного в класс TextQuery
int TextQuery::QueryResult::static_mem = 1024;
```

Поиск имен в области видимости вложенного класса

Во вложенном классе выполняются обычные правила поиска имен (см. раздел 7.4.1). Конечно, поскольку вложенный класс — это вложенная область видимости, для поиска у него есть дополнительные области видимости в содержащем классе. Такое вложение областей видимости объясняет, почему переменная-член `line_no` не определялась во вложенной версии класса `QueryResult`. Первоначальный класс `QueryResult` определял этот член для того, чтобы его собственные члены могли избежать необходимости записи `TextQuery::line_no`. После вложения определения класса результатов в класс `TextQuery` такое определение типа больше не нужно. Вложенный класс `QueryResult` может обратиться к переменной `line_no` без указания, что она определена в классе `TextQuery`.

Как уже упоминалось, вложенный класс — это тип-член содержащего его класса. Члены содержащего класса могут использовать имена вложенного класса таким же образом, как и любой другой тип-член. Поскольку класс `QueryResult` вложен в класс `TextQuery`, функция-член `query()` класса `TextQuery` может обращаться к имени `QueryResult` непосредственно:

```
// тип возвращаемого значения должен указать, что
класс QueryResult
// теперь вложенный
TextQuery::QueryResult
TextQuery::query( const string &sought) const {
    // если искомое значение не найдено, возвратить
указатель на этот
    // набор
    static     shared_ptr<set<line_no>>      nodata( new
set<line_no> );
    // во избежания добавления слов к wm использовать
поиск, а не
    // индексирование!
    auto loc = wm.find( sought );
    if ( loc == wm.end() )
        return QueryResult( sought, nodata, file ); // не
найдено
    else
        return QueryResult( sought, loc->second, file );
```

```
}
```

Как обычно, тип возвращаемого значения не находится в области видимости класса (см. раздел 7.4), поэтому сразу было обращено внимание на то, что функция возвращает значение типа `TextQuery::QueryResult`. Но в теле функции к типу `QueryResult` можно обращаться непосредственно, как это сделано в операторах `return`.

Вложенные и содержащие классы независимы

Несмотря на то что вложенный класс определяется в пределах содержащего его класса, важно понимать, что никакой связи между объектами содержащего класса и объектами его вложенного класса (классов) нет. Объект вложенного типа только содержит члены, определенные во вложенном типе. Точно так же у объекта содержащего класса есть только те члены, которые определяются содержащим классом. Он не содержит переменные-члены любых вложенных классов.

Конкретней, второй оператор `return` в функции-члене `TextQuery::query()` использует переменные-члены объекта класса `TextQuery`, для которого была выполнена функция `query()`, инициализирующая объект класса `QueryResult`:

```
return QueryResult( sought, loc->second, file);
```

Эти члены используются для создания возвращаемого объекта класса `QueryResult`, поскольку он не содержит члены содержащего его класса.

Упражнения раздела 19.5

Упражнение 19.20. Вложите собственный класс `QueryResult` в класс `TextQuery` и повторно запустите написанную в разделе 12.3.2 программу, использующую класс `TextQuery`.

19.6. Класс объединения, экономящий место

Класс *объединения* (*union*) — это специальный вид класса. У него может быть несколько переменных-членов, но в любой момент времени значение может быть только у одного из членов. Когда присваивается значение одному из членов класса объединения, все остальные члены становятся неопределенными. Объем хранилища, резервируемого для объединения, достаточен для содержания наибольшей переменной-члена. Подобно любому классу, класс объединения определяет новый тип.

Некоторые, но не все средства класса объединения применяются одинаково. У класса объединения не может быть члена, являющегося ссылкой, но у него могут быть члены большинства других типов, включая, согласно новому стандарту, типы классов с конструкторами или деструкторами. Объединение может использовать спецификаторы доступа, чтобы сделать члены открытыми закрытыми, или защищенными. По умолчанию, как и у структуры, члены объединения являются открытыми.

Класс объединения может определять функции-члены, включая конструкторы и деструкторы. Но объединения не могут происходить от другого класса и не могут быть использованы как базовый класс. В результате у объединения не может быть виртуальных функций.

Определение объединения

Объединения позволяют создать набор взаимоисключающих значений, которые могут иметь разные типы. Предположим, например, что существует процесс, в ходе которого обрабатываются различные виды числовых или символьных данных. Для хранения этих значений можно было бы использовать следующее объединение.

```
// объект типа Token способен содержать один член,
имеющий любой из
// следующих типов
union Token {
    // члены по умолчанию открыты
    char cval;
    int ival;
    double dval;
};
```

Определение объединения начинается с ключевого слова *union*, за которым следует имя объединения (не обязательно) и набор его членов,

заключенный в фигурные скобки. Этот код определяет объединение по имени `Token`, способное содержать значение типа `char`, `int` или `double`.

Использование объединения

Имя объединения — это имя типа. Подобно встроенным типам, по умолчанию объединения не инициализированы. Объединение можно явно инициализировать таким же образом, как и агрегатные классы (см. раздел 7.5.5), — при помощи инициализаторов, заключенных в фигурные скобки:

```
Token first_token = { 'a' }; // инициализирует член  
cval
```

```
Token last_token; // не инициализированный  
объект Token
```

```
Token *pt = new Token; // указатель на не  
инициализированный  
// объект Token
```

Если инициализатор есть, он используется для инициализации первого члена. Следовательно, инициализация объединения `first_token` присваивает значение его члену `cval`.

К членам объекта типа объединения обращаются при помощи обычных операторов доступа к члену:

```
last_token.cval = 'z';  
pt->iVal = 42;
```

Присвоение значения переменной-члену объекта объединения делает другие его переменные-члены неопределенными. В результате при использовании объединения следует всегда знать, какое именно значение в настоящее время хранится в нем. В зависимости от типов членов возвращение или присвоение хранимого в объединении значения при помощи неправильной переменной-члена может привести к аварийному отказу или неправильному поведению программы.

Анонимные объединения

Анонимное объединение (*anonymous union*) — это безымянное объединение, не содержащее объявлений между закрывающей фигурной скобкой, завершающей его тело, и точкой с запятой, завершающей определение объединения (см. раздел 2.6.1). При определении анонимного объединения компилятор автоматически создает безымянный объект только что определенного типа объединения:

```
union { // анонимное объединение
```

```
char cval;
int ival;
double dval;
} ; // определяет безымянный объект, к членам
которого можно обращаться
// непосредственно
cval = 'c'; // присваивает новое значение
безымянному, анонимному
// объекту объединения
ival = 42; // теперь этот объект содержит значение
42
```

Члены анонимного объединения непосредственно доступны в той области видимости, где определено анонимное объединение.



У анонимного объединения не может быть закрытых или защищенных членов, кроме того, оно не может определять функции-члены.

Объединения с членами типа класса



По прежним стандартам языка C++ у объединений не могло быть членов типа класса, которые определяли бы собственные конструкторы или функции-члены управления копированием. По новому стандарту это ограничение снято. Однако объединения с членами, способными определять собственные конструкторы и (или) функции-члены управления копированием, куда сложней в применении, чем объединения только с членами встроенного типа.

Если у объединения есть члены только встроенного типа, для изменения содержащегося в нем значения можно использовать обычное присвоение. С объединениями, у которых есть члены нетривиальных типов, все не так просто. При присвоении или замене значения члена объединения типа класса следует создать или удалить этот член соответственно: при присвоении объединению значения типа класса следует запустить конструктор для типа данного элемента, а при замене — запустить его деструктор.

Если у объединения есть члены только встроенного типа, компилятор сам синтезирует почленные версии стандартного конструктора и функций-членов управления копированием. Но для объединений, у которых есть член типа класса, определяющего собственный стандартный конструктор или функция-член управления копированием, это не так. Если тип члена объединения определяет одну из этих функций-членов, компилятор синтезирует соответствующий член объединения как удаленный (см. раздел 13.1.6).

Например, класс `string` определяет все пять функций-членов управления копированием, а также стандартный конструктор. Если объединение будет содержать строку и не определит ее собственный стандартный конструктор или одну из функций-членов управления копированием, то компилятор синтезирует эту недостающую функцию как удаленную. Если у класса будет член типа объединения, у которого есть удаленная функция-член управления копированием, то соответствующая функция (функции) управления копированием самого класса также будет удалена.

Использование класса для управления членами объединения

Из-за сложностей создания и удаления членов типа класса такие объединения обычно встраивают в другой классе. Таким образом, класс получает возможность управлять состоянием при передаче из и в элемент типа класса. В качестве примера добавим в объединение член класса `string`. Определим объединение как анонимное и сделаем его членом класса `Token`. Класс `Token` будет управлять членами объединения.

Для отслеживания вида значения хранимого объединением обычно определяют отдельный объект, *дискриминант* (*discriminant*). Дискриминант позволяет различать значения, которые может содержать объединение. Для синхронизации объединения и его дискриминанта сделаем дискриминант также членом класса `Token`. Для отслеживания состояния члена объединения класс определит член типа перечисления (см. раздел 19.3).

Единственными определяемыми классом функциями будут стандартный конструктор, функции-члены управления копированием и ряд операторов присвоения, способных присваивать значение одного из типов объединения члену другого:

```
class Token {  
public:  
    // функции управления копированием необходимы
```

потому, что у класса

```

    // есть объединение с членом типа string
    // определение конструктора перемещения и
оператора присваивания при
    // перемещении остается в качестве
самостоятельного упражнения
Token( ): tok(INT), ival{0} { }
Token( const Token &t): tok(t.tok) { copyUnion( t);
}
Token &operator=( const Token& );
// если объединение содержит строку, ее придется
удалять;
// см. раздел 19.1.2
~Token( ) { if (tok == STR) sval.~string( ); }
// операторы присвоения для установки разных
членов объединения
Token &operator=( const std::string& );
Token &operator=( char );
Token &operator=( int );
Token &operator=( double );
private:
enum { INT, CHAR, DBL, STR} tok; // дискриминант
union { // анонимное объединение
    char cval;
    int ival;
    double dval;
    std::string sval;
}; // у каждого объекта класса Token есть
безымянный член типа этого
// безымянного объединения
// проверить дискриминант и скопировать член
объединения, как надо
void copyUnion( const Token& );
};
```

Класс определяет вложенное, безымянное перечисление с не ограниченной областью видимости (см. раздел 19.3), используемое как тип члена *tok*. Член *tok* определен после закрывающей фигурной скобки и перед точкой с запятой, завершающей определение перечисления, которое определяет *tok*, как имеющий тип этого безымянного перечисления (см.

раздел 2.6.1).

Член `tok` будет использован как дискриминант. Когда объединение содержит значение типа `int`, член `tok` будет содержать значение `INT`; если объединение содержит значение типа `string`, то член `tok` содержит значение `STR` и т.д.

Стандартный конструктор инициализирует дискриминант и член объединения как содержащие значение 0 типа `int`.

Поскольку объединение содержит член, класс которого обладает деструктором, следует определить собственный деструктор, чтобы (условно) удалять член типа `string`. В отличие от обычных членов типа класса, члены типа класса, являющиеся частью объединения, не удаляются автоматически. У деструктора нет никакого способа узнать, значение какого типа хранит объединение. Таким образом, он не может знать, какой из членов следует удалить.

Поэтому деструктор проверяет, не содержит ли удаляемый объект строку. Если это так, то деструктор явно вызывает деструктор класса `string` (см. раздел 19.1.2) для освобождения используемой памяти. Если объединение содержит значение любого из встроенных типов, то деструктор не делает ничего.

Управление дискриминанта и удаление строки

Операторы присвоения устанавливают значение переменной `tok` и присваивают соответствующий член объединения. Подобно деструктору, эти функции-члены должны условно удалять строку, прежде чем присваивать новое значение объединению:

```
Token &Token::operator=(int i) {
    if (tok == STR) sval.~string(); // если это
строка, освободить ее
    ival = i; // присвоить соответствующий член
    tok = INT; // обновить дискриминант
    return *this;
}
```

Если текущим значением объединения является строка, ее следует освободить прежде, чем присвоить объединению новое значение. Для этого используется деструктор класса `string`.

Как только член типа `string` освобождается, предоставленное значение присваивается члену, тип которого соответствует типу параметра оператора. В данном случае параметр имеет тип `int`, поэтому он

присваивается `ival`. Затем обновляется дискриминант и осуществляется выход.

Операторы присвоения для типов `double` и `char` ведут себя, как и версия для типа `int`, их определение остается в качестве самостоятельного упражнения. Версия для типа `string` отличается от других, поскольку она должна управлять переходом от типа `string` и к нему:

```
Token &Token::operator=(const std::string &s) {
    if (tok == STR) // если строка уже содержится,
        просто присвоить новую
        sval = s;
    else
        new(&sval) string(s); // в противном случае
        создать строку
    tok = STR; // обновить дискриминант
    return *this;
}
```

В данном случае, если объединение уже содержит строку, можно использовать обычный оператор присвоения класса `string`, чтобы предоставить новое значение существующей строке. В противном случае не будет никакого объекта класса `string` для вызова его оператора присвоения. Вместо этого придется создать строку в памяти, которая содержит объединение. Для создания строки в области, где располагается `sval`, используется размещающий оператор `new` (см. раздел 19.1.2). Стока инициализируется копией строкового параметра, затем обновляется дискриминант и осуществляется выход.

Управление членами объединения, требующее управления копированием

Подобно специфическим для типа операторам присвоения, конструктор копий и операторы присвоения должны проверять дискриминант, чтобы знать, как копировать переданное значение. Для выполнения этих действий определим функцию-член `copyUnion()`.

Когда происходит вызов функции `copyUnion()` из конструктора копий, член объединения будет инициализирован значением по умолчанию, означая, что будет инициализирован первый член объединения. Поскольку строка не является первым элементом, вполне очевидно, что объединение содержит не строку. Оператор присвоения должен учитывать возможность того, что объединение уже содержит

строку. Отработаем этот случай непосредственно в операторе присвоения. Таким образом, если параметр функции `copyUnion()` содержит строку, она должна создать собственную строку:

```
void Token::copyUnion( const Token &t) {  
    switch (t.tok) {  
        case Token::INT: ival = t.ival; break;  
        case Token::CHAR: cval = t.cval; break;  
        case Token::DBL: dval = t.dval; break;  
        // для копирования строки создать ее, используя  
размещающий  
        // оператор new; см. раздел 19.1.2  
        case Token::STR: new( &sval) string( t.sval); break;  
    }  
}
```

Для проверки дискриминанта эта функция использует оператор `switch` (см. раздел 5.3.2). Значения встроенных типов просто присваиваются соответствующему члену; если копируемый член имеет тип `string`, он создается.

Оператор присвоения должен отработать три возможности для члена типа `string`: левый и правый операнды являются строками; ни один из операндов не является строкой; один, но не оба операнда являются строкой:

```
Token &Token::operator=( const Token &t) {  
    // если этот объект содержит строку, а t нет,  
прежнюю строку следует  
    // освободить  
    if (tok == STR && t.tok != STR) sval.~string();  
    if (tok == STR && t.tok == STR)  
        sval = t.sval; // нет необходимости создавать  
новую строку  
    else  
        copyUnion(t); // создать строку, если t.tok  
содержит STR  
    tok = t.tok;  
    return *this;  
}
```

Если объединение в левом операнде содержит строку, а объединение в правом — нет, то сначала следует освободить прежнюю старую строку,

прежде чем присваивать новое значение члену объединения. Если оба объединения содержат строку, для копирования можно использовать обычный оператор присвоения класса `string`. В противном случае происходит вызов функции `copyUnion()`, осуществляющей присвоение. В функции `copyUnion()`, если правый operand — строка, создается новая строка в члене объединения левого операнда. Если ни один из operandов не будет строкой, то достаточно обычного присвоения.

Упражнения раздела 19.6

Упражнение 19.21. Напишите собственную версию класса `Token`.

Упражнение 19.22. Добавьте в класс `Token` член типа `Sales_data`.

Упражнение 19.23. Добавьте в класс `Token` конструктор перемещения и присвоения.

Упражнение 19.24. Объясните, что происходит при присвоении объекта класса `Token` самому себе.

Упражнение 19.25. Напишите операторы присвоения, получающие значения каждого типа в объединении.

19.7. Локальные классы

Класс, определенный в теле функции, называют *локальным классом* (*local class*). Локальный класс определяет тип, видимый только в той области видимости, в которой он определен. В отличие от вложенных классов, члены локального класса жестко ограничены.



Все члены локального класса, включая функции, должны быть полностью определены в теле класса. В результате локальные классы гораздо менее полезны, чем вложенные.

На практике требование полностью определять члены в самом классе, существенно ограничивает сложность, а следовательно, и возможности функций-членов локального класса. Функции локальных классов редко имеют размер, превышающий несколько строк кода. Более длинный код функций труднее прочитать и понять.

Кроме того, в локальном классе нельзя объявлять статические переменные-члены, поскольку нет никакого способа определить их.

Локальные классы не могут использовать переменные из области видимости функции

Локальный класс может обращаться далеко не ко всем именам из окружающей области видимости. Он может обращаться только к именам типов, статических переменных (см. раздел 6.1.1) и перечислений, определенных в окружающей локальной области видимости. Локальный класс не может использовать обычные локальные переменные той функции, в которой определен класс:

```
int a, val;
void foo(int val) {
    static int si;
    enum Loc { a = 1024, b }; // Bar локальна для foo
    struct Bar {
        Loc locVal; // ok: используется локальное имя
типа
        int barVal;
        void fooBar( Loc l = a) // ok: аргумент по
умолчанию Loc::a
    {
        barVal = val; // ошибка: val локален для foo
        barVal = ::val; // ok: используется глобальный
объект
        barVal = si; // ok: используется статический
локальный объект
        locVal = b; // ok: используется перечислитель
    }
}; // ...
}
```

К локальным классам применимы обычные правила доступа

Содержащая функция не имеет никаких специальных прав доступа к закрытым членам локального класса. Безусловно, локальный класс вполне может сделать содержащую функцию дружественной. Как правило, локальный класс определяет свои члены как открытые. Та часть программы, которая может обращаться к локальному классу, весьма ограничена. Локальный класс сосредоточен (инкапсулирован) в своей локальной области видимости. Дальнейшая инкапсуляция,

подразумевающая скрытие информации, безусловно, является излишней.

Поиск имен в локальном классе

Поиск имен в теле локального класса осуществляется таким же образом, как и у остальных классов. Имена, используемые в объявлениях членов класса, должны быть объявлены в области видимости до их применения. Имена, используемые в определениях членов, могут располагаться в любой части области видимости локального класса. Поиск имен, не найденных среди членов класса, осуществляется сначала в содержащей локальной области видимости, а затем вне области видимости, заключающей саму функцию.

Вложенные локальные классы

Вполне возможно вложить класс в локальный класс. В данном случае определение вложенного класса может располагаться вне тела локального класса. Однако вложенный класс следует определить в той же локальной области видимости, в которой определен локальный класс:

```
void foo() {  
    class Bar {  
        public:  
            // ...  
            class Nested; // объявление класса Nested  
    };  
    // определение класса Nested  
    class Bar::Nested {  
        // ...  
    };  
}
```

Как обычно, при определении члена вне класса следует указать область видимости имени. Следовательно, определение `Bar::Nested` означает класс `Nested`, определенный в пределах класса `Bar`.

Класс, вложенный в локальный класс, сам является локальным классом, со всеми соответствующими ограничениями. Все члены вложенного класса должны быть определены в теле самого вложенного класса.

19.8. Возможности, снижающие переносимость

Для поддержки низкоуровневого программирования язык C++ определяет набор средств, применение которых снижает переносимость приложений. *Непереносимое* (nonportable) средство специфично для определенных машин. Использующие такие средства программы зачастую требуют переделки кода при переносе с одной машины на другую. Одной из причин невозможности переноса является тот факт, что размеры арифметических типов на разных машинах разные (см. раздел 2.1.1).

В этом разделе рассматриваются два дополнительных средства, снижающих переносимость, унаследованных языком C++ от языка C: речь идет о битовых полях и спецификаторе `volatile`. Также будут рассмотрены директивы компоновки, которые тоже снижают переносимость.

19.8.1. Битовые поля

Класс может определить (нестатическую) переменную-член как *битовое поле* (bit-field). Битовое поле хранит определенное количество битов. Обычно битовые поля используются при необходимости передать двоичные данные другой программе или аппаратному устройству.



Расположение в памяти битовых полей зависит от конкретной машины.

У битового поля должны быть целочисленный тип или тип перечисления (см. раздел 19.3). Для битового поля обычно используют беззнаковый тип, поскольку поведение битового поля знакового типа зависит от реализации. Чтобы объявить член класса битовым полем, после его имени располагают двоеточие и константное выражение, указывающее количество битов:

```
typedef unsigned int Bit;
class File {
    Bit mode: 2;           // mode имеет 2 бита
    Bit modified: 1;       // modified имеет 1 бит
    Bit prot_owner: 3;     // prot_owner имеет 3 бита
```

```

Bit prot_group: 3; // prot_group имеет 3 бита
Bit prot_world: 3; // prot_world имеет 3 бита
// функции и переменные-члены класса File
public:
    // режимы файла определены как восьмеричные
    // литералы; см. р. 2.1.3
    enum modes { READ = 01, WRITE = 02, EXECUTE = 03
};

File &open( modes );
void close();
void write();
bool isRead() const;
void setWrite();
}

```

Битовое поле mode имеет размер в два бита, битовое поле modified — только один, а другие — по три бита. Битовые поля, определенные в последовательном порядке в теле класса, если это возможно, упаковываются в смежных битах того же целого числа. Таким образом достигается уплотнение хранилища. Например, пять битовых полей в приведенном выше объявлении будут сохранены в одной переменной типа `unsigned int`, ассоциированной с первым битовым полем mode. Способ упаковки битов в целое число зависит от машины.

К битовому полю не может быть применен оператор обращения к адресу (`&`), поэтому не может быть никаких указателей на битовые поля классов.



ВНИМАНИЕ

Для битовых полей обычно лучше подходит беззнаковый тип. Поведение битовых полей, хранимых в переменной знакового типа, определяет конкретная реализация.

Использование битовых полей

К битовым полям обращаются так же, как и к другим переменным-членам класса:

```

void File::write() {
    modified = 1;
    // ...
}

```

```

    }
    void File::close() {
        if (modified)
            // ... сохранить содержимое
    }
}

```

Для манипулирования битовыми полями с несколькими битами обычно используют встроенные побитовые операторы (см. раздел 4.8):

```

File &File::open( File::modes m) {
    mode |= READ; // установить бит READ по умолчанию
    // другая обработка
    if (m & WRITE) // если открыто для чтения и записи
        // процесс открытия файла в режиме чтения/записи
    return *this;
}

```

Классы, определяющие члены битовых полей, обычно определяют также набор встраиваемых функций-членов для проверки и установки значений битовых полей:

```

inline bool File::isRead() const { return mode & READ; }
inline void File::setWrite() { mode |= WRITE; }

```

19.8.2. Спецификатор `volatile`



ВНИМАНИЕ

Смысл спецификатора `volatile` полностью зависит от конкретной машины и может быть выяснен только в документации компилятора. При переносе на новые машины или компиляторы программы, использующие спецификатор `volatile`, обычно приходится переделывать.

Программы, которым приходится работать непосредственно с аппаратными средствами, зачастую имеют элементы данных, значением которых управляют процессы, не контролируемые самой программой. Например, программа могла бы содержать переменную, значение которой изменяет системный таймер. Такой объект должен быть объявлен со спецификатором `volatile`, тогда его значение может быть изменено способами, не контролируемыми или не обнаруживаемыми компилятором.

Ключевое слово `volatile` — это приказ компилятору не выполнять оптимизацию для таких объектов.

Спецификатор `volatile` используется аналогично спецификатору `const`, т.е. как дополнительный модификатор типа:

```
volatile int display_register; // значение int
может изменяться
volatile Task *curr_task; // curr_task указывает
на объект volatile
volatile int iax[max_size]; // каждый элемент в iax
volatile volatile
Screen bitmapBuf; // каждый член
bitmapBuf volatile
```

Между спецификаторами типа `const` и `volatile` нет никакой взаимосвязи. Тип может быть и `const`, и `volatile`, тогда у него есть оба качества.

Точно так же класс может определить константные функции-члены, а может и асинхронно-изменяемые (`volatile`). Только асинхронно-изменяемые функции-члены могут быть вызваны асинхронно-изменяемым (`volatile`) объектом.

Взаимодействие указателей со спецификатором `const` описано в разделе 2.4.2. Аналогичное взаимодействие существует между указателями и спецификатором `volatile`. Можно объявлять асинхронно-изменяемые указатели на объекты, указатели на асинхронно-изменяемые объекты и асинхронно-изменяемые указатели на асинхронно-изменяемые объекты.

```
volatile int v; // v - асинхронно-изменяемый
объект типа int
int *volatile vip; // vip - асинхронно-изменяемый
указатель на тип int
volatile int *ivp; // ivp - указатель на
асинхронно-изменяемый тип int
// vivp - асинхронно-изменяемый указатель на
асинхронно-изменяемый
// объект типа int
volatile int *volatile vivp;
int *ip = &v; // ошибка: нужен указатель на
volatile
*ivp = &v; // ok: ivp - указатель на volatile
vivp = &v; // ok: vivp - volatile указатель на
```

volatile

Подобно константам, адрес асинхронно-изменяемого объекта можно присвоить (или скопировать указатель на асинхронно-изменяемый тип) только асинхронно-изменяемому указателю. При инициализации ссылки на асинхронно-изменяемый объект следует использовать только асинхронно-изменяемые ссылки.

Синтезируемые функции управления копированием не применимы к асинхронно-изменяемым объектам

Между константными и асинхронно-изменяемыми объектами есть одно важное различие: для инициализации и присвоения асинхронно-изменяемых объектов не применимы синтезируемые версии операторов присвоения, копирования и перемещения. Синтезируемые функции-члены управления копированием получают параметры, типами которых являются константные ссылки на класс. Однако асинхронно-изменяемый объект не может быть передан при помощи обычной или константной ссылки.

Если класс должен обеспечить копирование, перемещение или присвоение асинхронно-изменяемых объектов в (или из) асинхронно-изменяемый операнд, в нем следует определить его собственные версии операторов копирования и перемещения. Например, объявив параметры как ссылки `const` и `volatile`, можно обеспечить копирование или присвоение из любого вида типа `Foo`:

```
class Foo {  
public:  
    Foo( const volatile Foo& );      // копирование из  
объекта volatile  
    // присвоение объекта volatile обычному объекту  
    Foo& operator=( volatile const Foo& );  
    // присвоение объекта volatile объекту volatile  
    Foo& operator=( volatile const Foo& ) volatile;  
    // остальная часть класса Foo  
};
```

Хотя для объектов `volatile` вполне можно определить функции копирования и присвоения, возникает вполне резонный вопрос: имеет ли смысл копировать объект `volatile`? Ответ зависит от причины использования такого объекта в конкретной программе.

19.8.3. Директивы компоновки: `extern "C"`

Иногда в программах C++ необходимо применять функции, написанные на другом языке программирования. Как правило, это язык C. Подобно любому имени, имя функции, написанной на другом языке, следует объявить. Это объявление должно указать тип возвращаемого значения и список параметров. Компилятор проверяет обращения к внешним функциям на другом языке точно так же, как и обращения к обычным функциям языка C++. Однако для вызова функций, написанных на других языках, компилятор обычно вынужден создавать иной код. Чтобы указать язык для функций, написанных на языке, отличном от C++, используются *директивы компоновки* (linkage directive).



Комбинация кода C++ с кодом, написанным на любом другом языке, включая язык C, требует доступа к компилятору этого языка, совместимому с вашим компилятором C++.

Объявление функций, написанных на языке, отличном от C++

Директива компоновки может существовать в двух формах: одиночной и составной. Директивы компоновки не могут располагаться в определении класса или функции. Некоторые директивы компоновки должны присутствовать в каждом объявлении функции.

В качестве примера рассмотрим некоторые из функций языка C, объявленные в заголовке `cstdlib`:

```
// гипотетические директивы компоновки, которые
 могли бы
 // присутствовать в заголовке C++ <cstring>
 // одиночная директива компоновки
 extern "C" size_t strlen(const char *);
 // составная директива компоновки
 extern "C" {
     int strcmp(const char*, const char*);
     char *strcat(char*, const char*);
 }
```

Первая форма состоит из ключевого слова `extern`, сопровождаемого строковым литералом и "обычным" объявлением функции.

Строчный литерал указывает язык, на котором написана функция.

Используемый компилятор обязан поддерживать директивы компоновки для языка С. Компилятор может поддерживать директивы компоновки и для других языков, например `extern "Ada"`, `extern "FORTRAN"` и т.д.

Директивы компоновки и заголовки

Та же директива компоновки может быть применена к нескольким функциям одновременно. Для этого их объявления заключают в фигурные скобки после директивы компоновки. Эти фигурные скобки служат для группировки объявлений, к которым применяется директива компоновки. Эти фигурные скобки игнорируются, а имена функций, объявленных в их пределах, видимы, как будто функции были объявлены вне фигурных скобок.

Составная форма объявления применима ко всему файлу заголовка. Например, заголовок `cstring` языка С++ может выглядеть следующим образом.

```
// составная директива компоновки
extern "C" {
    #include <string.h>      // функции языка C,
    манипулирующие строками
                                // в стиле C
}
```

Когда директива `#include` заключена в фигурные скобки составной директивы компоновки, все объявления обычных функций в файле заголовка будут восприняты как написанные на языке, указанном в директиве компоновки. Директивы компоновки допускают вложенность, т.е. если заголовок содержит функцию с директивой компоновки, на данную функцию это не повлияет.



Функции, унаследованные языком С++ от языка С, могут быть определены как функции языка С, но это не является обязательным условием для каждой реализации языка С++.

Указатели на функции, объявленные в директиве `extern "C"`

Язык, на котором написана функция, является частью ее типа. Чтобы

объявить указатель на функцию, написанную на другом языке программирования, следует использовать директиву компоновки. Кроме того, указатели на функции, написанные на других языках, следует объявлять с той же директивой компоновки, что и у самой функции:

```
// pf указывает на функцию С, возвращающую void и
// получающую int
```

```
extern "C" void (*pf) (int);
```

Когда указатель pf используется для вызова функции, созданный при компиляции код подразумевает, что происходит обращение к функции С.

Тип указателя на функцию С не совпадает с типом указателя на функцию С++. Указатель на функцию С не может быть инициализирован (или присвоен) значением указателя на функцию С++ (и наоборот). Как и при любом другом несовпадении типов, попытка присвоения указателя с другой директивой компоновки приведет к ошибке:

```
void (*pf1)(int); // указатель на
функцию С++
```

```
extern "C" void (*pf2)(int); // указатель на
функцию С
```

```
pf1 = pf2; // ошибка: pf1 и pf2 имеют разные типы
```



ВНИМАНИЕ

Некоторые компиляторы С++ могут допускать присвоение, приведенное выше, хотя, строго говоря, оно некорректно.

Директивы компоновки применимы ко всем объявлениюм

Директива компоновки, использованная для функции, применяется также и к любым указателям на нее, используемым как тип возвращаемого значения или параметр.

```
// f1() - функция С, ее параметр также является
// указателем на функцию С
```

```
extern "C" void f1(void(*)(int));
```

Это объявление свидетельствует о том, что f1() является функцией языка С, которая не возвращает никаких значений. Она имеет один параметр в виде указателя на функцию, которая ничего не возвращает и получает один параметр типа int. Эта директива компоновки применяется как к самой функции f1(), так и к указателю на нее. Когда происходит вызов функции f1(), ей необходимо передать имя функции С или

указатель на нее.

Поскольку директива компоновки применяется ко всем функциям в объявлении, для передачи функции C++ указателя на функцию С необходимо использовать определение типа (см. раздел 2.5.1):

```
// FC - указатель на функцию С
extern "C" typedef void FC( int );
// f2 - функция С++, параметром которой является
указатель на функцию С
void f2( FC * );
```

Экспорт функций, созданных на языке С++, в другой язык

Используя директиву компоновки в определении функции, написанной на языке С++, эту функцию можно сделать доступной для программы, написанной на другом языке.

```
// функция calc() может быть вызвана из программы
на языке С
extern "C" double calc( double dparam ) { /* ... */ }
```

Код, создаваемый компилятором для этой функции, будет соответствовать указанному языку.

Следует заметить, что типы параметров и возвращаемого значения в функциях для разных языков зачастую ограничены. Например, почти наверняка нельзя написать функцию, которая передает объекты нетривиального класса С++ программе на языке С. Программа С не будет знать о конструкторах, деструкторах или других специфических для класса операциях.

Поддержка препроцессора при компоновке на языке С

Чтобы позволить компилировать тот же файл исходного кода на языке С или С++, при компиляции на языке С++ препроцессор автоматически определяет имя `_cplusplus` (два символа подчеркивания). Используя эту переменную, при компиляции на С++ можно условно включить код, компилируемый только на С++:

```
#ifdef __cplusplus
// ok: компилируется только в С++
extern "C"
#endif
int strcmp( const char*, const char* );
```

Перегруженные функции и директивы компоновки

Взаимодействие директив компоновки и перегрузки функций зависит от конкретного языка. Если язык поддерживает перегрузку функций, то компилятор, обрабатывая директивы компоновки для того языка, вероятней всего, выполнит ее.

Язык С не поддерживает перегрузку функций, поэтому нет ничего удивительного в том, что директива компоновки языка С может быть определена только для одной из функций в наборе перегруженных функций:

```
// ошибка: в директиве extern "C" указаны две
одноименные функции
extern "C" void print(const char*);
extern "C" void print(int);
```

Если одна из функций в наборе перегруженных функций является функцией языка С, все остальные функции должны быть функциями С++:

```
class SmallInt { /* ... */ };
class BigNum { /* ... */ };
// функция С может быть вызвана из программ С и С++
// версия функции С++, перегружающая предыдущую
функцию, может быть
// вызвана только из программ на языке С++
extern "C" double calc(double);
extern SmallInt calc(const SmallInt&);
extern BigNum calc(const BigNum&);
```

Версия функции calc() для языка С может быть вызвана как из программ на языке С, так и из программ на языке С++. Дополнительные функции с параметрами типа класса могут быть вызваны только из программ на языке С++, причем порядок объявления не имеет значения.

Упражнения раздела 19.8.3

Упражнение 19.26. Объясните эти объявления и укажите, допустимы ли они:

```
extern "C" int compute(int *, int);
extern "C" double compute(double *, double);
```

Резюме

Язык С++ предоставляет несколько специализированных средств, предназначенных для решения ряда специфических проблем.

Некоторым приложениям требуется взять под свой контроль

распределение памяти. Это можно сделать, определив собственные версии (в классе или глобально) библиотечных функций `operator new()` и `operator delete()`. Если приложение определяет собственные версии этих функций, выражения `new` и `delete` будут использовать соответствующую версию, определенную приложением.

Некоторым программам необходимо непосредственно выяснить динамический тип объекта во время выполнения. Идентификация типов времени выполнения (Run-Time Type Identification — RTTI) предоставляет поддержку этого вида программирования на уровне языка. RTTI применима только к тем классам, которые обладают виртуальными функциями; информация о типах без виртуальных функций также доступна, но она соответствует статическому типу.

При определении указателя на член класса в состав его типа должен также входить тот класс, на член которого указывает указатель. Указатель на член класса может быть связан с членом любого объекта того же класса. При обращении к значению указателя на член класса необходимо указать объект, о члене которого идет речь.

В языке C++ определено несколько дополнительных составных типов.

- Вложенные классы, которые определены в области видимости другого класса. Такие классы зачастую применяют для реализации содержащего класса.

- Объединения — это специальный вид класса, объект которого может содержать только простые переменные-члены. В любой момент времени объект такого типа может содержать значение только в одной из его переменных-членов. Как правило, объединения входят в состав другого класса.

- Локальные классы представляют собой очень простые классы, определенные локально в функции. Все члены локального класса должны быть определены в его теле. Для локального класса недопустимы статические переменные-члены.

Язык C++ предоставляет также несколько средств, ухудшающих переносимость программ. Сюда относятся битовые поля, спецификатор `volatile`, упрощающий взаимодействие с аппаратными средствами, и директивы компоновки, упрощающие взаимодействие с программами, написанными на других языках.

Термины

Анонимное объединение (anonymous union). Безымянное объединение,

которое не применимо для создания объекта. Члены анонимного объединения являются членами окружающей области видимости. Такие объединения не могут иметь ни функций-членов, ни закрытых или защищенных членов.

Битовое поле (bit-field). Целочисленный член класса, определяющий количество резервируемых для него битов. Битовые поля, определенные в классе последовательно, могут быть упакованы в обычное целочисленное значение.

Вложенный класс (nested class). Класс, определенный в другом классе. Вложенный класс определен в окружающей области видимости: имена вложенных классов должны быть уникальны в области видимости того класса, в котором они определены, но могут повторяться в областях видимости вне содержащего класса. Доступ к вложенному классу извне содержащего класса предполагает применение оператора области видимости, позволяющего указать область (области) видимости, в которую вложен класс.

Вложенный тип (nested type). Синоним вложенного класса.

Директива компоновки (linkage directive). Механизм, позволяющий вызвать в программе на языке C++ функции, написанные на другом языке. Вызов функций С должны поддерживать все компиляторы языка C++. Поддержка других языков зависит от конкретного компилятора.

Идентификация типов времени выполнения (run-time type identification). Языковые и библиотечные средства, позволяющие выяснить динамический тип ссылки или указателя во время выполнения. Операторы RTTI, typeid и dynamic_cast, обеспечивают возвращение динамического типа только для ссылок и указателей на классы с виртуальными функциями. Будучи примененными к другим типам, они возвращают статический тип ссылки или указателя.

Локальный класс (local class). Класс, определенный в функции. Локальный класс видим только в той функции, в которой он определен. Все его члены должны быть определены в теле класса. Он не может иметь статических членов. Локальные члены класса не могут обращаться к локальным переменным, определенным в содержащей функции. Однако они могут использовать имена типов, статические переменные и перечисления, определенные в содержащей функции.

Непереносимый (nonportable). Специфические для конкретных машин средства, которые могут потребовать изменений при переносе программы на другую машину или компилятор.

Объединение (union). Подобный классу составной тип, в котором

может быть определено несколько переменных-членов, однако значение в каждый момент времени может иметь только один из них. Объединения могут иметь функции-члены, включая конструкторы и деструкторы, но они не могут быть использованы в качестве базового класса. По новому стандарту у объединений могут быть члены-типы, определяющие собственные функции-члены управления копированием. Такие объединения получают удаленные функции управления копированием, если они не определяют соответствующие функции управления копированием.

Оператор `delete`. Библиотечная функция, освобождающая динамическую память без контроля типов, зарезервированную оператором `new`. Библиотечный оператор `delete[]` освобождает память, задействованную массивом, который был зарезервирован оператором `new[]`.

Оператор `dynamic_cast`. Осуществляет приведение типа базового класса к типу производного с проверкой. В базовом классе должна быть определена по крайней мере одна виртуальная функция. Оператор проверяет динамический тип объекта, с которым связана ссылка или указатель. Приведение осуществляется только тогда, когда тип объекта совпадает с типом приведения или является типом, производным от него. В противном случае возвращается нулевой указатель (при приведении указателя) или исключение (при приведении ссылки).

Оператор `typeid`. Унарный оператор, получающий выражение и возвращающий ссылку на объект библиотечного типа `type_info`, описывающего тип полученного выражения. Когда выражение является объектом класса, имеющего виртуальные функции, оператор возвращает динамический тип. Если типом является ссылка, указатель или другой тип, в котором не определены виртуальные функции, будет возвращен его статический тип. Выражение не вычисляется.

Перечисление (*enumeration*). Тип, группирующий набор именованных целочисленных констант.

Перечисление с не ограниченной областью видимости (*unscoped enumeration*). Перечисление, перечислители которого доступны в окружающей области видимости.

Перечисление с ограниченной областью видимости (*scoped enumeration*). Перечисление нового вида, в котором перечислитель не доступен непосредственно в окружающей области видимости.

Перечислитель (*enumerator*). Именованный член перечисления.

Каждый перечислитель инициализируется константным целочисленным значением. Перечислители могут быть использованы там, где необходимы целочисленные константные выражения.

Размещающий оператор new (placement new). Форма оператора new, создающая объект в указанной области памяти. Память он не резервирует, а область, предназначенную для объекта, указывает получаемый аргумент. Представляет собой низкоуровневый аналог функции-члена construct() класса allocator.

Спецификатор volatile. Спецификатор типа, указывающий компилятору на то, что значение переменной данного типа может быть изменено извне программы. Это запрещает компилятору осуществлять некоторые виды оптимизации кода.

Тип type_info. Библиотечный тип, возвращаемый оператором typeid. Класс type_info жестко зависит от конкретной машины, однако любая библиотека должна определять класс type_info как содержащий функцию-член name(), возвращающую символьную строку, представляющую имя типа. Объекты класса type_info не могут быть скопированы, перемещены или присвоены.

Указатель на член класса (pointer to member). Инкапсулирует тип класса, а также тип элемента, на который он указывает. Определение указателя на член класса должно содержать имя класса, а также тип элемента (элементов), на который он может указывать.

```
T C:: * pmem = &C:: { member};
```

Это выражение определяет указатель pmem, который способен указывать на члены класса по имени C, которые имеют тип T, и инициализирует его адресом члена класса C по имени member. Перед обращением к значению такого указателя он должен быть предварительно связан с объектом или указателем класса C.

```
classobj. * pmem;
classptr->* pmem;
```

Обращение к члену member объекта classobj или указателя classptr.

Функция free(). Низкоуровневая функция освобождения памяти, определенная в заголовке cstdlib. Функция free() может использоваться для освобождения только той памяти, которая зарезервирована функцией malloc().

Функция malloc(). Низкоуровневая функция резервирования памяти, определенная в заголовке cstdlib. Зарезервированную функцией

`malloc()` память следует освобождать функцией `free()`.

Шаблон`mem_fn`. Библиотечный шаблон класса, создающий вызываемый объект из переданного указателя на функцию-член.

Приложения

Приложение А

Библиотека

Это приложение содержит дополнительные сведения об алгоритмах и разделе случайных чисел библиотеки. В начале приведена табл. А.1, содержащая имена и заголовки стандартной библиотеки, упоминаемые в книге.

В главе 10 были использованы некоторые из наиболее популярных алгоритмов и описана архитектура, лежащая в их основе. В данном приложении перечислены все алгоритмы, упорядоченные по выполняемым ими операциям.

В разделе 17.4 была описана библиотечная архитектура для случайных чисел, а также приведены примеры использования распределений нескольких типов. Библиотека определяет несколько процессоров случайного числа и двадцать распределений различных видов. В этом приложении перечислены все процессоры и распределения.

А.1. Имена и заголовки стандартной библиотеки

В программах этой книги директивы `#include`, необходимые для их компиляции, практически нигде не приводились. Для удобства читателей в табл. А.1 перечислены все использованные в программах книги библиотечные имена и заголовки, в которых они определены.

Таблица А.1. Имена и заголовки стандартной библиотеки

Имя	Заголовок
abort	<cstdlib>
accumulate	<numeric>
allocator	<memory>
array	<array>
auto_ptr	<memory>
back_inserter	<iterator>
bad_alloc	<new>
bad_array_new_length	<new>
bad_cast	<typeinfo>
begin	<iterator>
bernoulli_distribution	<random>
bind	<functional>
bitset	<bitset>
boolalpha	<iostream>
cerr	<iostream>
cin	<iostream>
cmatch	<regex>
copy	<algorithm>
count	<algorithm>
count_if	<algorithm>
cout	<iostream>
cref	<functional>
csub_match	<regex>
dec	<iostream>
default_float_engine	<iostream>

default_random_engine	<random>
deque	<deque>
domain_error	<stdexcept>
end	<iterator>
endl	<iostream>
ends	<iostream>
equal_range	<algorithm>
exception	<exception>
fill	<algorithm>
fill_n	<algorithm>
find	<algorithm>
find_end	<algorithm>
find_first_of	<algorithm>
find_if	<algorithm>
fixed	<iostream>
flush	<iostream>
for_each	<algorithm>
forward	<utility>
forward_list	<forward_list>
free	cstdlib
front_inserter	<iterator>
fstream	<fstream>
function	<functional>
get	<tuple>
getline	<string>
greater	<functional>
hash	<functional>
hex	<iostream>
hexfloat	<iostream>
ifstream	<fstream>
initializer_list	<initializer_list>
inserter	<iterator>
internal	<iostream>
ios_base	<ios_base>

isalpha	<cctype>
islower	<cctype>
isprint	<cctype>
ispunct	<cctype>
isspace	<cctype>
istream	<iostream>
istream_iterator	<iterator>
istringstream	<sstream>
isupper	<cctype>
left	<iostream>
less	<functional>
less_equal	<functional>
list	<list>
logic_error	<stdexcept>
lower_bound	<algorithm>
lround	<cmath>
make_move_iterator	<iterator>
make_pair	<utility>
make_shared	<memory>
make_tuple	<tuple>
malloc	cstdlib
map	<map>
max	<algorithm>
max_element	<algorithm>
mem_fn	<functional>
min	<algorithm>
move	<utility>
multimap	<map>
multiset	<set>
negate	<functional>
noboolalpha	<iostream>
normal_distribution	<random>
noshowbase	<iostream>
noshowpoint	<iostream>

noskipws	<iostream>
not1	<functional>
nothrow	<new>
nothrow_t	<new>
nounitbuf	<iostream>
nouppercase	<iostream>
nth_element	<algorithm>
oct	<iostream>
ofstream	<fstream>
ostream	<iostream>
ostream_iterator	<iterator>
ostringstream	<sstream>
out_of_range	<stdexcept>
pair	<utility>
partial_sort	<algorithm>
placeholders	<functional>
placeholders::_1	<functional>
plus	<functional>
priority_queue	<queue>
ptrdiff_t	<cstddef>
queue	<queue>
rand	<random>
random_device	<random>
range_error	<stdexcept>
ref	<functional>
regex	<regex>
regex_constants	<regex>
regex_error	<regex>
regex_match	<regex>
regex_replace	<regex>
regex_search	<regex>
remove_pointer	<type_traits>
remove_reference	<type_traits>
replace	<algorithm>

replace_copy	<algorithm>
reverse_iterator	<iterator>
right	<iostream>
runtime_error	<stdexcept>
scientific	<iostream>
set	<set>
set_difference	<algorithm>
set_intersection	<algorithm>
set_union	<algorithm>
setfill	<iomanip>
setprecision	<iomanip>
setw	<iomanip>
shared_ptr	<memory>
showbase	<iostream>
showpoint	<iostream>
size_t	<cstddef>
skipws	<iostream>
smatch	<regex>
sort	<algorithm>
sqrt	<cmath>
sregex_iterator	<regex>
ssub_match	<regex>
stable_sort	<algorithm>
stack	<stack>
stoi	<string>
strcmp	<cstring>
strcpy	<cstring>
string	<string>
stringstream	<sstream>
strlen	<cstring>
strncpy	<cstring>
strtod	<string>
swap	<utility>
terminate	<exception>

time	<ctime>
tolower	<cctype>
toupper	<cctype>
transform	<algorithm>
tuple	<tuple>
tuple_element	<tuple>
tuple_size	<tuple>
type_info	<typeinfo>
unexpected	<exception>
uniform_int_distribution	<random>
uniform_real_distribution	<random>
uninitialized_copy	<memory>
uninitialized_fill	<memory>
unique	<algorithm>
unique_copy	<algorithm>
unique_ptr	<memory>
unitbuf	<iostream>
unordered_map	<unordered_map>
unordered_multimap	<unordered_map>
unordered_multiset	<unordered_set>
unordered_set	<unordered_set>
upper_bound	<algorithm>
uppercase	<iostream>
vector	<vector>
weak_ptr	<memory>

А.2. Краткий обзор алгоритмов

В библиотеке определено более 100 алгоритмов. Чтобы научиться их использовать, следует понять структуру, а не запоминать подробности применения каждого из них. Лежащая в их основе архитектура описана в главе 10, а в этом разделе описан каждый из алгоритмов.

- `beg` и `end` — итераторы, обозначающие диапазон элементов (см. раздел 9.2.1). Почти все алгоритмы работают с последовательностями, обозначенными итераторами `beg` и `end`.

- `beg2` — итератор, обозначающий начало второй исходной последовательности. Если итератор `end2` присутствует, он обозначает конец второй последовательности. Если итератора `end2` нет, подразумевается, что обозначенная итератором `beg2` последовательность такого же размера, что и исходная, обозначенная итераторами `beg` и `end`. Типы итераторов `beg` и `beg2` не обязаны совпадать. Но должна существовать возможность применить указанную операцию или заданный вызываемый объект к элементам этих двух последовательностей.

- `dest` — итератор, обозначающий назначение. Последовательность назначения должна быть способна содержать столько элементов, сколько необходимо для исходной последовательности.

- `unaryPred` и `binaryPred` — унарные и бинарные предикаты (см. раздел 10.3.1), возвращающие применимый в условии тип и получающие соответственно один и два аргумента, являющиеся элементами исходного диапазона.

- `comp` — бинарный предикат, отвечающий требованиям упорядочивания по ключу в ассоциативном контейнере (см. раздел 11.2.2).

- `unaryOp` и `binaryOp` — вызываемые объекты (см. раздел 10.3.2), которые могут быть вызваны с одним и двумя аргументами из исходного диапазона соответственно.

А.2.1. Алгоритмы поиска объекта

Эти алгоритмы осуществляют поиск в исходной последовательности заданного значения или последовательности значений.

Каждый алгоритм предоставляет две перегруженных версии. Первая версия для сравнения элементов использует оператор равенства (`==`) базового типа, а вторая использует предоставленные пользователем предикаты `unaryPred` или `binaryPred`.

Простой алгоритм поиска

Для поиска этим алгоритмам требуются *итераторы ввода*.

```
find( beg, end, val)
find_if( beg, end, unaryPred)
find_if_not( beg, end, unaryPred)
count( beg, end, val)
count_if( beg, end, unaryPred)
```

Функция `find()` возвращает итератор на первый элемент в исходном диапазоне, равный значению `val`. Функция `find_if()` возвращает итератор на первый элемент, для которого выполняется предикат `unaryPred`. Функция `find_if_not()` возвращает итератор на первый элемент, для которого предикат `unaryPred` возвращает значение `false`. Все три функции возвращают итератор `end`, если искомый элемент не существует.

Функция `count()` возвращает количество вхождений значения `val`. Функция `count_if()` подсчитает количество элементов, для которых предикат `unaryPred` возвращает значение `true`.

```
all_of( beg, end, unaryPred)
any_of( beg, end, unaryPred)
none_of( beg, end, unaryPred)
```

Возвращают логическое значение, указывающее, выполняется ли предикат `unaryPred` для всех элементов, какого-нибудь элемента или ни одного элемента соответственно. Если последовательность пуста, функция `any_of()` возвращает значение `false`, а функции `all_of()` и `none_of()` — `true`.

Алгоритм поиска одного из нескольких значений

Этим алгоритмам требуется *прямые итераторы*. Они ищут в исходной последовательности повторяющиеся элементы.

```
adjacent_find( beg, end)
adjacent_find( beg, end, binaryPred)
```

Возвращает итератор на первую пару смежных совпадающих элементов. Возвращает итератор `end`, если смежных совпадающих элементов нет.

```
search_n( beg, end, count, val)
search_n( beg, end, count, val, binaryPred)
```

Возвращает итератор на начало внутренней последовательности из `count` равных элементов. Возвращает итератор `end`, если такой

внутренней последовательности не существует.

Алгоритм поиска последовательности

За исключением алгоритма `find_first_of()` этим алгоритмам требуются две пары *прямых итераторов*. Для обозначения первой своей последовательности алгоритм `find_first_of()` использует *итераторы ввода* и *прямые итераторы* для второй. Эти алгоритмы ищут последовательность, а не одиночный элемент.

```
search( beg1, end1, beg2, end2)
search( beg1, end1, beg2, end2, binaryPred)
```

Возвращает итератор на первую позицию исходного диапазона, с которой начинается искомая последовательность. Возвращает итератор `end1`, если искомая последовательность не найдена.

```
find_first_of( beg1, end1, beg2, end2)
find_first_of( beg1, end1, beg2, end2, binaryPred)
```

Возвращает итератор на первое вхождение в первом диапазоне любого элемента из второго диапазона. Возвращает итератор `end1`, если искомое соответствие отсутствует.

```
find_end( beg1, end1, beg2, end2)
find_end( beg1, end1, beg2, end2, binaryPred)
```

Подобен алгоритму `search()`, но возвращает итератор на последнюю позицию в исходном диапазоне, в которой второй диапазон встречается как внутренняя последовательность. Возвращает итератор `end1`, если вторая последовательность пуста или не найдена.

A.2.2. Другие алгоритмы, осуществляющие только чтение

Для первых двух аргументов этим алгоритмам требуются *итераторы ввода*.

Алгоритмы `equal()` и `mismatch()` получают также дополнительный *итератор ввода*, обозначающий начало второго диапазона. Они также предоставляют две перегруженных версии. Первая версия для сравнения элементов использует оператор равенства (`==`) базового типа, а вторая сравнивает элементы используя предоставленный пользователем предикат `unaryPred` или `binaryPred`.

```
for_each( beg, end, unaryOp)
```

Вызываемый объект (см. раздел 10.3.2) `unaryOp` применяется к каждому элементу в исходном диапазоне. Возвращаемое значение объекта

`unaryOp` (если оно есть) игнорируется. Если итераторы позволяют запись в элементы при помощи оператора обращения к значению, то вызываемый объект `unaryOp` способен изменять элементы.

```
mismatch( beg1, end1, beg2)
mismatch( beg1, end1, beg2, binaryPred)
```

Сравнивает элементы в двух последовательностях. Возвращает пару (см. раздел 11.2.3) итераторов, обозначающих первые элементы в каждой не совпадающей последовательности. Если все элементы соответствуют друг другу, первый итератор возвращенной пары окажется равным `end1`, а итератор `beg2` — смещению, равному размеру первой последовательности.

```
equal( beg1, end1, beg2)
equal( beg1, end1, beg2, binaryPred)
```

Выявляет равенство двух последовательностей. Возвращает значение `true`, если каждый элемент в исходном диапазоне равен соответствующему элементу последовательности, начинающейся с позиции `beg2`.

A.2.3. Алгоритмы бинарного поиска

Хотя эти алгоритмы можно использовать с прямыми итераторами, они обладают специализированными версиями, которые работают с итераторами прямого доступа и выполняются гораздо быстрей.

Этим алгоритмам требуется *прямые итераторы*, но они оптимизированы так, что выполняются намного быстрее, если вызываются с *итераторами прямого доступа*. С технической точки зрения, независимо от типа итератора, эти алгоритмы выполняют логарифмическое количество сравнений. Но при использовании с прямыми итераторами они должны выполнить линейное количество операций с итераторами для перебора элементов последовательности.

Эти алгоритмы требуют, чтобы элементы в исходной последовательности уже были упорядочены. Эти алгоритмы ведут себя подобно одноименным функциям-членам ассоциативных контейнеров (см. раздел 11.3.5).

Алгоритмы `equal_range()`, `lower_bound()` и `upper_bound()` возвращают итераторы на позиции последовательности, куда мог бы быть вставлен заданный элемент при сохранении существующего порядка в последовательности. Если элемент больше всех остальных в последовательности, то возвращаемый итератор будет итератором после

конца.

Каждый алгоритм предоставлен в двух версиях: первая использует для проверки элементов оператор меньше ($<$) типа элемента, а вторая использует заданную функцию сравнения. В следующих алгоритмах "x меньше, чем y" означает, что выражения $x < y$ и $\text{comp}(x, y)$ истинны:

```
lower_bound( beg, end, val)
lower_bound( beg, end, val, comp)
```

Возвращает итератор, обозначающий первый элемент, значение которого больше или равно значению val , или итератор end , если такого элемента нет.

```
upper_bound( beg, end, val)
upper_bound( beg, end, val, comp)
```

Возвращает итератор, обозначающий первый элемент, значение которого меньше значения val , или итератор end , если такого элемента нет.

```
equal_range( beg, end, val)
equal_range( beg, end, val, comp)
```

Возвращает пару (см. раздел 11.2.3), член `first` которой является итератором, возвращаемым функцией `lower_bound()`, а член `second` — итератором, возвращаемым функцией `upper_bound()`.

```
binary_search( beg, end, val)
binary_search( beg, end, val, comp)
```

Возвращает логическое значение, свидетельствующее о наличии в последовательности элемента, значение которого равно val . Два значения, x и y , считаются равными, если x не меньше y и y не меньше x .

A.2.4. Алгоритмы записи в элементы контейнера

Запись в элементы контейнера осуществляется многими алгоритмами. Эти алгоритмы могут отличаться видом итераторов, используемых для обозначения их исходной последовательности, а также тем, осуществляют ли они запись в элементы исходного диапазона или указанного результирующего диапазона.

Алгоритмы, которые записывают, но не читают значения элементов

Для обозначения назначения этим алгоритмам требуются *итераторы вывода*. Версии `_n` получают второй определяющий количество аргумент и записывают заданный набор элементов по назначению.

```
fill( beg, end, val)
fill_n( dest, cnt, val)
generate( beg, end, Gen)
generate_n( dest, cnt, Gen)
```

Присваивают новое значение каждому элементу исходной последовательности. Алгоритм `fill()` присваивает значение `val`; алгоритм `generate()` выполняет объект генератора `Gen`. Генератор — это вызываемый объект (см. раздел 10.3.2), возвращающий при каждом вызове разные значения. Алгоритмы `fill()` и `generate()` возвращают тип `void`. Версии `_n` возвращают итератор на позицию непосредственно после последнего элемента, записанного в последовательность назначения.

Алгоритмы записи с итераторами ввода

Каждый из этих алгоритмов читает исходную последовательность и пишет последовательность вывода. Они требуют, чтобы `dest` был *итератором вывода*, а итераторы, обозначающие исходный диапазон, должны быть *итераторами ввода*.

```
copy( beg, end, dest)
copy_if( beg, end, dest, unaryPred)
copy_n( beg, n, dest)
```

Копирует из исходного диапазона последовательности, обозначенные итератором `dest`. Алгоритм `copy()` копирует все элементы, а алгоритм `copy_if()` копирует те из них, для которых предикат `unaryPred` истин, а алгоритм `copy_n()` копирует первые `n` элементов. У исходной последовательности должно быть по крайней мере `n` элементов.

```
move( beg, end, dest)
```

Вызов функции `std::move()` (см. раздел 13.6.1) для каждого элемента в исходной последовательности позволяет переместить этот элемент в последовательность, начиная с итератора `dest`.

```
transform( beg, end, dest, unaryOp)
transform( beg, end, beg2, dest, binaryOp)
```

Вызывает заданную операцию и пишет ее результат в `dest`. Первая версия применяет унарную операцию к каждому элементу в исходном диапазоне. Вторая применяет бинарную операцию к элементам этих двух исходных последовательностей.

```
replace_copy( beg, end, dest, old_val, new_val)
replace_copy_if( beg, end, dest, unaryPred, new_val)
```

Копируют каждый элемент в `dest`, заменяя определенные элементы

значением `new_val`. Первая версия заменяет элементы == `old_val`, а вторая версия — элементы, удовлетворяющие предикату `unaryPred`.

```
merge( beg1, end1, beg2, end2, dest)
merge( beg1, end1, beg2, end2, dest, comp)
```

Сортирует обе исходные последовательности. Записывает в `dest` объединенную последовательность. Первая версия сравнивает элементы при помощи оператора `<`; а вторая использует предоставленный оператор сравнения.

Алгоритмы записи с прямыми итераторами

Этим алгоритмам требуются *прямые итераторы*, поскольку они пишут в элементы своих исходных последовательностей. Итераторы должны предоставлять доступ для записи в элементы.

```
iter_swap( iter1, iter2)
swap_ranges( beg1, end1, beg2)
```

Заменяет элемент, обозначенный итератором `iter1`, элементом, обозначенным итератором `iter2`; или обменивает все элементы в исходном диапазоне с таковыми из второй последовательности, начиная с позиции `beg2`. Диапазоны не должны пересекаться. Алгоритм `iter_swap()` возвращает `void`; алгоритм `swap_ranges` возвращает итератор `beg2`, увеличенный так, чтобы обозначить элемент сразу после последнего обмененного.

```
replace( beg, end, old_val, new_val)
replace_if( beg, end, unaryPred, new_val)
```

Заменяет каждый элемент, соответствующий значению `new_val`. Первая версия использует для сравнения элементов со значением `old_val` оператор ==, а вторая заменяет те элементы, для которых истинен предикат `unaryPred`.

Алгоритмы записи с двунаправленными итераторами

Поскольку этим алгоритмам необходима способность вернуться назад в последовательности, они требуют *двунаправленных итераторов*.

```
copy_backward( beg, end, dest)
move_backward( beg, end, dest)
```

Копирует или перемещает элементы из исходного диапазона в заданный. В отличие от других алгоритмов, `dest` — итератор после конца для выходной последовательности (т.е. последовательность назначения закончится непосредственно *перед* `dest`). Последний элемент в исходном

диапазоне копируется или перемещается в последний элемент назначения, затем копируется (перемещается) предпоследний элемент и т.д. У элементов в последовательности назначения тот же порядок, что и в исходном диапазоне. Если диапазон пуст, возвращается итератор `dest`, в противном случае возвращается итератор на элемент, который был скопирован или перемещен из `*beg`.

```
inplace_merge(beg, mid, end)
inplace_merge(beg, mid, end, comp)
```

Объединяет две отсортированные внутренние последовательности из той же последовательности в единую, упорядоченную последовательность. Внутренние последовательности от `beg` до `mid` и от `mid` до `end` объединяются и записываются назад в первоначальную последовательность. Первая версия использует для сравнения элементов оператор `<`, а вторая версия использует заданную операцию сравнения. Возвращают `void`.

A.2.5. Алгоритмы сортировки и разделения

Алгоритмы сортировка и разделения предоставляют различные стратегии упорядочивания элементов последовательности.

Каждый алгоритм сортировки и разделения поддерживает стабильные и нестабильные версии (см. раздел 10.3.1). Стабильный алгоритм обеспечивает относительный порядок равных элементов. Стабильные алгоритмы выполняют больше работы, а следовательно, могут выполняться медленней и использовать больше памяти, чем нестабильные аналоги.

Алгоритмы разделения

Алгоритмы разделения делят элементы исходного диапазона на две группы. Первая группа состоит из элементов удовлетворяющих определенному предикату, а вторая — нет. Например, элементы последовательности можно разделить на основании четности их значений или на основании того, начинается ли слово с заглавной буквы, и так далее. Этим алгоритмам требуются *двунаправленные итераторы*.

```
is_partitioned(beg, end, unaryPred)
```

Возвращает значение `true`, если все элементы, удовлетворяющие предикату `unaryPred`, предшествуют тем, для которых предикат `unaryPred` возвращает значение `false`. Если последовательность пуста, также возвращается значение `true`.

```
partition_copy( beg, end, dest1, dest2, unaryPred)
```

Копирует в dest1 элементы, для которых истин предикат unaryPred, а остальные копирует в dest2. Возвращает пару (см. раздел 11.2.3) итераторов. Член first пары обозначает конец скопированных в dest1 элементов, а член second обозначает конец элементов, скопированных в dest2. Исходная последовательность не может налагаться ни на одну из результирующих последовательностей.

```
partition_point( beg, end, unaryPred)
```

Для разделения исходной последовательности используется предикат unaryPred. Возвращает итератор на элемент за последним, удовлетворяющим предикату unaryPred. Если возвращен итератор не end, то предикат unaryPred должен возвращать значение false для возвращенного итератора и для всех элементов, следующих за ним.

```
stable_partition( beg, end, unaryPred)
```

```
partition( beg, end, unaryPred)
```

Для разделения исходной последовательности используется предикат unaryPred. Элементы, для которых истин предикат unaryPred, помещаются в начало последовательности, а остальные в конец. Возвращает итератор на элемент за последним, удовлетворяющим предикату unaryPred, или итератор beg, если таких элементов нет.

Алгоритмы сортировки

Этим алгоритмам требуется *итераторы прямого доступа*. Каждый из алгоритмов сортировки предоставляется в двух перегруженных версиях. В одной из них для сравнения элементов используется оператор < типа элемента, а во второй предусмотрен дополнительный параметр для функции сравнения (см. раздел 11.2.2). Алгоритм partial_sort_copy() возвращает итератор получателя, а остальные возвращают void.

Алгоритмы partial_sort() и nth_element() выполняют частичную сортировку последовательности. Их используют в случае, когда в результате сортировки всей последовательности могут возникнуть проблемы. Поскольку эти операции являются менее трудоемкими, они выполняются быстрее, чем сортировка всего исходного диапазона.

```
sort( beg, end)
```

```
stable_sort( beg, end)
```

```
sort( beg, end, comp)
```

```
stable_sort( beg, end, comp)
```

Сортирует весь диапазон.

```
is_sorted( beg, end)
is_sorted( beg, end, comp)
is_sorted_until( beg, end)
is_sorted_until( beg, end, comp)
```

Алгоритм `is_sorted()` возвращает логическое значение, указывающее, сортируется ли вся исходная последовательность. Алгоритм `is_sorted_until()` находит самую длинную изначально отсортированную часть в исходной последовательности и возвращает итератор на позицию сразу после ее конца.

```
partial_sort( beg, mid, end)
partial_sort( beg, mid, end, comp)
```

Сортирует набор элементов, количество которых равно `mid - beg`. То есть если `mid - beg` равно 42, эта функция помещает элементы с самыми низкими значениями, в отсортированном порядке, в первые 42 позиции последовательности. После завершения работы алгоритма `partial_sort()` окажутся отсортированы элементы в диапазоне от `beg` и далее, но не включая `mid`. Ни один из элементов в отсортированном диапазоне не больше, чем любой из элементов в диапазоне после `mid`. Порядок неотсортированных элементов не определен.

```
partial_sort_copy( beg, end, destBeg, destEnd)
partial_sort_copy( beg, end, destBeg, destEnd, comp)
```

Сортирует элементы исходного диапазона и помещает их (в отсортированном порядке) в последовательность, указанную итераторами `destBeg` и `destEnd`. Если получающий диапазон имеет тот же размер или превосходит исходный, в него сохраняется весь исходный диапазон в отсортированном виде, начиная с позиции `destBeg`. Если размер получающего диапазона меньше, в него будет скопировано столько отсортированных элементов, сколько поместится.

Алгоритм возвращает итератор в получающем диапазоне, указывающий на следующий элемент после последнего отсортированного. Если получающая последовательность меньше исходного диапазона или равна ему по размеру, будет возвращен итератор `destEnd`.

```
nth_element( beg, nth, end)
nth_element( beg, nth, end, comp)
```

Аргумент `nth` должен быть итератором, указывающим на элемент в исходной последовательности. Обозначенный этим итератором элемент после выполнения алгоритма `nth_element` имеет значение, которое

находилось бы там после сортировки всей последовательности. Кроме того, элементы контейнера вокруг позиции `nth` также отсортированы: перед ней располагается значение меньше или равное значению в позиции `nth`, а после нее значение, большее или равное.

А.2.6. Общие функции изменения порядка

Некоторые алгоритмы переупорядочивают элементы исходной последовательности. Первые два, `remove()` и `unique()`, переупорядочивают последовательность так, чтобы элементы в первой части удовлетворяли некоему критерию. Они возвращают итератор, отмечающий конец этой подпоследовательности. Другие, например `reverse()`, `rotate()` и `random_shuffle()`, реорганизуют всю последовательность.

Базовые версии этих алгоритмов работают "на месте", т.е. они реорганизуют элементы непосредственно исходной последовательности. Три алгоритма изменения порядка предоставляют копирующие версии. Они записывают переупорядоченные значения в получающую последовательность, а не непосредственно в исходную. Для получающей последовательности этим алгоритмам требуются *итераторы вывода*.

Переупорядочивающие алгоритмы, использующие прямые итераторы

Эти алгоритмы переупорядочивают исходную последовательность. Им необходимы по крайней мере *прямые итераторы*.

```
remove( beg, end, val)
remove_if( beg, end, unaryPred)
remove_copy( beg, end, dest, val)
remove_copy_if( beg, end, dest, unaryPred)
```

"Удаляет" элементы из последовательности, записывая поверх них те элементы, которые должны быть сохранены. Удаляются те элементы, которые равны значению `val` или те, для которых предикат `unaryPred` вернул значение `true`. Возвращает итератор на следующий элемент после последнего удаленного.

```
unique( beg, end)
unique( beg, end, binaryPred)
unique_copy( beg, end, dest)
unique_copy_if( beg, end, dest, binaryPred)
```

Переупорядочивает последовательность так, чтобы смежные

совпадающие элементы были удалены при перезаписи. Возвращает итератор на следующий элемент после последнего уникального. Для проверки совпадения двух смежных элементов первая версия использует оператор `==`, а вторая — предикат.

```
rotate( beg, mid, end)
rotate_copy( beg, mid, end, dest)
```

"Поворачивает" элементы вокруг элемента, обозначенного итератором `mid`. Элемент, указанный итератором `mid`, становится первым элементом, затем идет последовательность от `mid+1` до `end` (но не включая его), далее следует диапазон от `beg` до `mid` (но не включая его). Возвращает итератор, обозначающий элемент, который первоначально был в `beg`.

Переупорядочивающие алгоритмы, использующие двунаправленные итераторы

Поскольку эти алгоритмы обрабатывают исходную последовательность в обратном порядке, им необходимы *двунаправленные итераторы*.

```
reverse( beg, end)
reverse_copy( beg, end, dest)
```

Меняет порядок элементов последовательности на обратный. Алгоритм `reverse()` возвращает тип `void`, а алгоритм `reverse_copy()` возвращает итератор, принимающей последовательности на элемент, который расположен за последним скопированным.

Переупорядочение алгоритмов с помощью итераторов прямого доступа

Поскольку эти алгоритмы реорганизуют элементы в произвольном порядке, им нужны *итераторы прямого доступа*.

```
random_shuffle( beg, end)
random_shuffle( beg, end, rand)
shuffle( beg, end, Uniform_rand)
```

Осуществляет перестановку элементов исходной последовательности в случайном порядке. Перетасовывает элементы в исходной последовательности. Вторая версия получает вызываемый объект, получающий положительное целочисленное значение и возвращающий случайное целое число в диапазоне от нуля до за данного значения с равномерным распределением. Третий аргумент должен отвечать требованиям равномерного генератора случайных чисел (см. раздел 17.4). Все три версии возвращают тип `void`.

A.2.7. Алгоритмы перестановки

Алгоритмы перестановки осуществляют лексикографические перестановки последовательности. Эти алгоритмы переупорядочивают элементы так, чтобы получить лексикографически следующую или предыдущую перестановку заданной последовательности. Они возвращают логическое значение, означающее, была ли осуществлена следующая или предыдущая перестановка.

Чтобы лучше понять смысл следующей или предыдущей лексикографической перестановки, рассмотрим такую последовательность из трех символов: $a b c$. У этой последовательности есть шесть возможных вариантов перестановки: $a b c$, $a c b$, $b a c$, $b c a$, $c a b$ и $c b a$. Эти варианты перестановки перечислены в лексикографическом порядке на основании оператора "меньше". Таким образом, вариант перестановки $a b c$ будет первым, поскольку его первый элемент меньше или равен первому элементу любого другого варианта перестановки, а ее второй элемент меньше, чем у любого другого варианта с тем же первым элементом. Точно так же $a c b$ — следующий вариант перестановки, поскольку он начинается с символа a , который меньше первого элемента любого из остальных вариантов перестановки. Варианты перестановки, начинающиеся с b , располагаются перед таковыми, начинаящимися с c .

Для каждого описанного выше варианта перестановки можно выяснить, какой из них должен располагаться прежде, а какие после него. Например, варианте перестановки $b c a$ можно сказать, что предыдущим для нее будет вариант $b a c$, а следующим — $c a b$. Для варианта $a b c$ нет предыдущего, а для варианта $c b a$ — последующего варианта перестановки.

Эти алгоритмы подразумевают, что элементы в последовательности уникальны. Таким образом, алгоритмы подразумевают, что никакие два элемента последовательности не имеют одинакового значения.

Для осуществления перестановки нужна возможность перебора последовательности вперед и назад, поэтому им требуются *двунаправленные операторы*.

```
is_permutation( beg1, end1, beg2)
is_permutation( beg1, end1, beg2, binaryPred)
```

Алгоритмы возвращают значение `true`, если во второй последовательности есть вариант перестановки с тем же набором элементов, что и в первой последовательности, для которой элементы в варианте перестановки и в исходной последовательности равны. Первая

версия сравнивает элементы, используя оператор `==`; вторая использует заданный предикат `binaryPred`.

```
next_permutation( beg, end)
next_permutation( beg, end, comp)
```

Если последовательность уже находится в последнем варианте перестановки, алгоритм `next_permutation()` переупорядочивает последовательность так, чтобы она соответствовала самой младшей версии, и возвращает значение `false`. В противном случае последовательность преобразуется в следующий вариант перестановки и возвращает значение `true`. Первая версия использует для сравнения элементов оператор `<` типа элемента, а вторая — указанную функцию сравнения.

```
prev_permutation( beg, end)
prev_permutation( beg, end, comp)
```

Подобен алгоритму `next_permutation()`, но преобразует последовательность в предыдущую версию перестановки. Если текущая версия является самой младшей, переупорядочивает последовательность в самую старшую и возвращает значение `false`.

A.2.8. Алгоритмы набора для отсортированных последовательностей

Алгоритмы набора реализуют присущие набору операции, применяемые для отсортированной последовательности. Не следует путать эти алгоритмы с функциями библиотечного контейнера `set` (набор). Они обеспечивают присущее набору поведение на базе обычного последовательного контейнера (например, `vector`, `list` и т.д.) или другой последовательности (например, потока ввода).

Поскольку эти алгоритмы обрабатывают элементы последовательно, им требуются *итераторы ввода*. За исключением алгоритма `includes` всем им необходим *итератор вывода*. Алгоритмы возвращают итератор `dest`, увеличенный так, чтобы указывать на следующий элемент после последнего записанного.

Каждый алгоритм предоставлен в двух формах: использующей для сравнения элементов оператор `<` или функцию сравнения.

```
includes( beg, end, beg2, end2)
includes( beg, end, beg2, end2, comp)
```

Возвращает значение `true`, если каждый элемент во второй

последовательности содержится в исходной последовательности. В противном случае возвращает значение `false`.

```
set_union( beg, end, beg2, end2, dest)
set_union( beg, end, beg2, end2, dest, comp)
```

Создает сортируемую последовательность элементов, которые находятся в обеих последовательностях. Элементы, которые находятся в обеих последовательностях, записываются в указанную итератором `dest` результирующую последовательность в одном экземпляре.

```
set_intersection( beg, end, beg2, end2, dest)
set_intersection( beg, end, beg2, end2, dest, comp)
```

Создает сортируемую последовательность элементов, представленных в обеих последовательностях. Результат сохраняется в последовательности, указанной итератором `dest`.

```
set_difference( beg, end, beg2, end2, dest)
set_difference( beg, end, beg2, end2, dest, comp)
```

Создает сортируемую последовательность элементов, представленных в первом контейнере, но не во втором.

```
set_symmetric_difference( beg, end, beg2, end2,
                          dest)
set_symmetric_difference( beg, end, beg2, end2,
                          dest, comp)
```

Создает сортируемую последовательность элементов, представленных в любом из контейнеров, но не в обоих контейнерах.

A.2.9. Минимальные и максимальные значения

Эти алгоритмы используют при сравнении либо оператор `<` для типа элемента, либо заданную функцию сравнения. Алгоритмы первой группы работают со значениями, а не с последовательностями. Алгоритмы второй группы получают последовательность, обозначенную *итераторами ввода*.

```
min( val1, val2)
min( val1, val2, comp)
min( init_list)
min( init_list, comp)
max( val1, val2)
max( val1, val2, comp)
max( init_list)
max( init_list, comp)
```

Эти алгоритмы возвращают минимум или максимум значений `val1` и

`val2` либо значений из списка `initializer_list`. Тип аргументов должен точно совпадать. Аргументы и тип возвращаемого значения являются ссылками на константы, а значит, объекты не копируются.

```
minmax( val1, val2)
minmax( val1, val2, comp)
minmax( init_list)
minmax( init_list, comp)
```

Возвращают пару (см. раздел 11.2.3), член `first` которой содержит меньшее из предоставленных значений, а член `second` — большее. Версия со списком `initializer_list` возвращает пару, член `first` которой содержит наименьшее значение в списке, а `second` — наибольшее.

```
min_element( beg, end)
min_element( beg, end, comp)
max_element( beg, end)
max_element( beg, end, comp)
minmax_element( beg, end)
minmax_element( beg, end, comp)
```

Алгоритмы `min_element()` и `max_element()` возвращают итераторы на наименьший и наибольший элементы в исходной последовательности соответственно. Алгоритм `minmax_element` возвращает пару, член `first` которой содержит наименьший элемент, а член `second` — наибольший.

Лексикографическое сравнение

Этот алгоритм сравнивает две последовательности в поисках первой неравной пары элементов. Используется либо оператор `<` типа элемента, либо заданная функция сравнения. Обе последовательности обозначаются итераторами ввода.

```
lexicographical_compare( beg1, end1, beg2, end2)
lexicographical_compare( beg1, end1, beg2, end2,
comp)
```

Алгоритм возвращает значение `true`, если первая последовательность лексикографически меньше второй. В противном случае возвращается значение `false`. Если одна последовательность короче второй и все ее элементы совпадают с соответствующими элементами более длинной последовательности, то более короткая последовательность лексикографически меньше. Если размер последовательностей совпадает и совпадают соответствующие элементы, то ни одна из них

лексикографически не меньше другой.

A.2.10. Числовые алгоритмы

Числовые алгоритмы определены в заголовке `numeric`. Этим алгоритмам требуются *итераторы ввода*; если алгоритм осуществляет запись в вывод, он использует *итератор вывода* для получателя.

```
accumulate( beg, end, init)
accumulate( beg, end, init, binaryOp)
```

Возвращает сумму всех значений в исходном диапазоне. Суммирование начинается с исходного значения, заданного параметром `init`. Тип возвращаемого значения задает тип параметра `init`. Первая версия использует оператор `+` типа элемента, а вторая — указанный бинарный оператор.

```
inner_product( beg1, end1, beg2, init)
inner_product( beg1, end1, beg2, init, binOp1,
binOp2)
```

Возвращает сумму элементов, полученных как произведение двух последовательностей. Обе последовательности обрабатываются совместно и элементы из каждой последовательности умножаются. Результат умножения суммируется. Исходное значение суммы определяет `init`. Тип `init` определяет тип возвращаемого значения.

Первая версия использует операторы умножения (`*`) и сложения (`+`) элементов. Вторая версия применяет заданные бинарные операторы, используя первый оператор вместо суммы и второй вместо умножения.

```
partial_sum( beg, end, dest)
partial_sum( beg, end, dest, binaryOp)
```

Пишет в `dest` новую последовательность, каждое значение элемента которой представляет собой сумму всех предыдущих элементов до (и включая) своей позиции в пределах исходного диапазона. Первая версия использует оператор `+` типа элемента, а вторая — заданный бинарный оператор. Возвращает итератор `dest`, увеличенный так, чтобы указывать на следующий элемент после последнего записанного.

```
adjacent_difference( beg, end, dest)
adjacent_difference( beg, end, dest, binaryOp)
```

Пишет в `dest` новую последовательность, каждое значение элемента которой, кроме первого, представляет собой разницу между текущими и предыдущим элементами. Первая версия использует оператор `-` тип

элемента, а вторая применяет заданный бинарный оператор.

```
iofa( beg, end, val)
```

Присваивает *val* первому элементу и осуществляет приращение *val*. Присваивает приращенное значение следующему элементу и снова осуществляет приращение *val*, а затем присваивает приращенное значение следующему элементу последовательности. Продолжает приращение *val* и присваивает новое значение последующему элементу в исходной последовательности.

A.3. Случайные числа

Библиотека определяет набор классов процессоров случайных чисел и адаптеров, использующих различные математические подходы для генерации псевдослучайных чисел. Библиотека определяет также набор шаблонов распределений, обеспечивающих распределение чисел согласно различным вероятностям. Имена классов процессоров и распределений соответствуют их математическим свойствам.

Подробности генерации чисел этими классами не рассматриваются в данном издании. Здесь перечислены типы процессоров и распределений, но чтобы лучше узнать, как их использовать, следует обратиться к другим ресурсам.

A.3.1. Распределение случайных чисел

За исключением распределения *bernoulli_distribution*, всегда генерирующего логические значения, типы распределений являются шаблонами. Каждый из этих шаблонов получает один параметр типа, задающий тип генерируемого распределением результата.

Классы распределений отличаются от других использованных ранее шаблонов класса, поскольку типы распределения налагают ограничения на пригодные для использования типы. Некоторые шаблоны распределения применяются для генерации только чисел с плавающей запятой; другие применяются для генерации только целых чисел.

В описаниях ниже для указания типа генерируемых шаблоном распределения чисел, например с плавающей запятой, используется формат *имя_шаблона*<*RealT*>. Для таких шаблонов вместо *RealT* можно использовать типы *float*, *double* или *long double*. Точно также вместо *IntT* можно использовать любой из встроенных целочисленных типов (*short*, *int*, *long*, *long long*, *unsigned short*, *unsigned*

`int`, `unsigned long` или `unsigned long long`), но не тип `bool` или `char`.

Шаблоны распределения определяют заданный по умолчанию параметр типа шаблона (см. раздел 17.4.2). Для целочисленных распределений по умолчанию принят тип `int`; для распределений, генерирующих числа с плавающей запятой, — тип `double`.

Конструкторы каждого вида распределения имеют специфические параметры. Некоторые из этих параметров определяют диапазон распределения. В отличие от диапазонов итераторов, эти диапазоны всегда являются инклюзивными (включающими крайние значения).

Равномерное распределение

```
uniform_int_distribution<IntT> u( m, n );
uniform_real_distribution<RealT> u( x, y );
```

Генерирует значения указанного типа в заданном инклюзивном диапазоне. Параметры `m` (или `x`) задают наименьшее число, которое может быть возвращено; а параметры `n` (или `y`) — наибольшее. По умолчанию `m` имеет значение 0, а `n` — максимально возможное значение, которое способен хранить объект типа `intT`. Параметр `x` по умолчанию имеет значение 0.0, а `y` — 1.0.

Распределение Бернулли

```
bernoulli_distribution b( p );
```

Возвращает значение `true` с вероятностью, заданной параметром `p`. По умолчанию параметр `p` имеет значение 0.5.

```
binomial_distribution<IntT> b( t, p );
```

Распределение вычисляется для выборочного размера, заданного целочисленным значением `t`, с вероятностью `p`; по умолчанию `t` имеет значение 1, а `p` — значение 0.5.

```
geometric_distribution<IntT> g( p );
```

Параметр `p` задает вероятность возвращения значения `true` и по умолчанию имеет значение 0.5.

```
negative_binomial_distribution<IntT> nb( k, p );
```

Целочисленное значение `k` приближается к решению с вероятностью успеха `p`. По умолчанию `k` имеет значение 1, а `p` — значение 0.5.

Распределение Пуассона

```
poisson_distribution<IntT> p( x );
```

Распределение относительно значения x типа `double`.
`exponential_distribution<RealT> e(lam);`
Лямбда `lam` — значение с плавающей точкой; по умолчанию `lam` имеет значение 1.0.

`gamma_distribution<RealT> g(a, b);`
Альфа (форма) `a` и бета (масштаб) `b`; оба по умолчанию имеют значение 1.0.

`weibull_distribution<RealT> w(a, b);`

Форма `a` и масштаб `b`; оба по умолчанию имеют значение 1.0.

`extreme_value_distribution<RealT> e(a, b);`

По умолчанию `a` имеет значение 0.0, а `b` — значение 1.0.

Нормальное распределение или распределение Гаусса

`normal_distribution<RealT> n(m, s);`

Параметр `m` — это математическое ожидание, а `s` — среднеквадратичное отклонение. По умолчанию `m` имеет значение 0.0, а `s` — значение 1.0.

`lognormal_distribution<RealT> ln(m, s);`

Параметр `m` — это математическое ожидание, а `s` — среднеквадратичное отклонение. По умолчанию `m` имеет значение 0.0, а `s` — значение 1.0.

`chi_squared_distribution<RealT> c(x);`

Параметр `x` — это степень свободы; по умолчанию имеет значение 1.0.

`cauchy_distribution<RealT> c(a, b);`

Область `a` по умолчанию имеет значение 0.0, а масштаб `b` — значение 1.0.

`fisher_f_distribution<RealT> f(m, n);`

`m` и `n` — степени свободы; оба по умолчанию имеют значения 1.

`student_t_distribution<RealT> s(n);`

`n` — степень свободы; значение по умолчанию — 1.

Выборочное распределение

`discrete_distribution<IntT> d(i, j);`

`discrete_distribution<IntT> d{ il};`

`i` и `j` — итераторы ввода последовательности коэффициентов; `il` — заключенный в скобки список коэффициентов. Коэффициенты должны допускать приведение к типу `double`.

`piecewise_constant_distribution<RealT> pc(b, e, w);`

`b, e` и `w` — итераторы ввода.

```
piecewise_linear_distribution<RealT> pl( b, e, w);  
b, e и w — итераторы ввода.
```

A.3.2. Процессоры случайных чисел

Библиотека определяет три класса, реализующих различные алгоритмы генерации случайных чисел. Библиотека определяет также три адаптера, модифицирующих созданную последовательность заданным процессором. Классы процессоров и адаптеров являются шаблонами. В отличие от параметров распределений, параметры процессоров сложны и требуют хорошего знания математического механизма, используемого конкретным процессором. Процессоры перечислены здесь только для того, чтобы читатель знал об их существовании, но подробно они в этой книге не рассматриваются.

Библиотека определяет также несколько типов, созданных на базе процессоров и адаптеров. Тип `default_random_engine` — это псевдоним типа для одного из классов процессоров, параметризованных переменными, предназначенными для повышения эффективности использования. Библиотека определяет также несколько классов, являющихся полностью специализированными версиями процессора или адаптера. Ниже приведены процессоры и их специализации, определенные библиотекой.

Tun default_random_engine

Псевдоним типа для одного из процессоров, подходящего для большинства задач.

Tun linear_congruential_engine

`minstd_rand0` — имеет множитель 16807, модуль 2147483647 и приращение 0.

`minstd_rand` — имеет множитель 48271, модуль 2147483647 и приращение 0.

Tun mersenne_twister_engine

`mt19937` — 32-разрядный беззнаковый генератор вихря Мерсенна.

`mt19937_64` — 64-разрядный беззнаковый генератор вихря Мерсенна.

Tun subtract_with_carry_engine

`ranlux24_base` — 32-разрядный беззнаковый генератор вычитания с переносом.

`ranlux48_base` — 64-разрядный беззнаковый генератор вычитания с переносом.

Tun discard_block_engine

Адаптер процессора, отбрасывающий результаты базового процессора. Параметризуется базовым процессором для размера используемого блока и размера использованных блоков.

`ranlux24` — использует процессор `ranlux24_base` с размером блока 223 и размером использованных блоков 23.

`ranlux48` — использует процессор `ranlux48_base` с размером блока 389 и размером использованных блоков 11.

Tun independent_bits_engine

Адаптер процессора, генерирующий числа с заданным количеством битов. Параметризован базовым процессором для использования количества битов, генерируемых в его результатах, и целочисленным беззнаковым типом, используемым для содержания созданных битов. Определяемое количество битов должно быть меньше количества цифр, которое может содержать заданный беззнаковый тип.

Tun shuffle_order_engine

Адаптер процессора, возвращающий те же числа, что и его базовый процессор, но в другой последовательности. Параметризован базовым процессором и количеством переставляемых элементов.

`knuth_b` — использует процессор `minstd_rand0` с размером таблицы 256.

1

На самом деле препроцессору. Компилятор получит готовый промежуточный файл, в состав которого войдет содержимое подключенной библиотеки. — *Примеч. ред.*

2

А также для временного отключения больших фрагментов кода при отладке. — *Примеч. ред.*

3

Согласно другой трактовке, исключение — это объект системного или пользовательского класса, создаваемого операционной системой или кодом программы в ответ на обстоятельства, либо не допускающие дальнейшего нормального выполнения программы, либо определенные пользователем. Обработка исключений в приложении позволяет корректно выйти из затруднительной ситуации. — *Примеч. ред.*

4

Здесь и везде в оригинале именно *adaptor*, а не *adapter*. — *Примеч. ред.*